Laboratorium 10

Rozwiązywanie równań różniczkowych

Piotr Magiera

10 czerwca 2022

1 Zadania

1. Dane jest równanie różniczkowe (zagadnienie początkowe):

$$y_0 + ycosx = sinxcosx,$$
$$y(0) = 0.$$

Znaleźć rozwiązanie metodą Rungego-Kutty i metodą Eulera. Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym:

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

2. 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:

$$y(x) = cosx - sinx + x.$$

2 Rozwiązania zadań

1. Metoda Rungego-Kutty opiera się na wzorze rekurencyjnym przedstawionym poniżej:

$$y_{n+1} = y_n + \Delta y_n,$$
$$\Delta y_n = (k_1 + 2k_2 + 3k_3 + k_4)/6,$$
$$k_1 = h * f(x_n, y_n),$$
$$k_2 = h * f(x_n + h/2, y_n + k_1/2),$$
$$k_3 = h * f(x_n + h/2, y_n + k_2/2),$$
$$k_4 = h * f(x_n + h, y_n + k_3).$$

Metoda Eulera to szczególny przypadek metody Rungego-Kutty, gdzie

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

Kod użyty do wygenerowania wyników tego zadania przedstawiono poniżej.

```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


# equation of type y' = f(x, y)
# y(x_0) = y_0
# x_n+1 = x_n + h
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


for x_val in [1, 2]:
    for n in [100, 1000, 10000, 100000, 1000000]:
        x, y = Runge_Kutta(n, x_val / n, 0, 0, f)
        print(f"Runge-Kutta: x={x:.2f}, error={abs(y - exact_solution(x))}")
        x, y = Euler(n, x_val / n, 0, 0, f)
        print(f"Euler: x={x:.2f}, error={abs(y - exact_solution(x))}")
```

Otrzymano następujące wyniki.

x	iterations	error Runge-Kutta	error Euler
1	100	$1.97 * 10^{-11}$	$7.41 * 10^{-4}$
1	1000	$1.99 * 10^{-15}$	$7.33 * 10^{-5}$
1	10000	$2.24 * 10^{-14}$	$7.33 * 10^{-7}$
1	100000	$6.19 * 10^{-13}$	$7.33 * 10^{-7}$
1	1000000	$3.07 * 10^{-12}$	$7.33 * 10^{-8}$
2	100	$4.04 * 10^{-10}$	$4.45 * 10^{-3}$
2	1000	$4.05 * 10^{-14}$	$4.44 * 10^{-4}$
2	10000	$2.35 * 10^{-14}$	$4.49 * 10^{-5}$
2	100000	$2.59 * 10^{-14}$	$4.49 * 10^{-6}$
2	1000000	$1.66 * 10^{-12}$	$4.49 * 10^{-7}$

Wnioski: metoda Rungego-Kutty jest metodą dużo dokładniejszą niż metoda Eulera, co było spodziewane. Wraz ze wzrostem liczby iteracji zwiększa się precyzja metody Eulera, natomiast metoda Rungego-Kutty jest na tyle dokładna, że nie jest to regułą - nie powinno to jednak stanowić problemu przy praktycznych zastosowaniach tej metody z uwagi na to, że i tak osiągana jest dokładność rzędu 10^{-10}. Można ponadto zauważyć, że precyzja zmniejsza się wraz ze wzrostem odległości argumentu x od argumentu, dla którego podana jest wartość funkcji w zagadnieniu początkowym (w naszym przypadku jest to $x_0 = 0$).

2. Metoda strzałów polega na zastąpieniu zagadnienia brzegowego postaci

$$y'' = f(x, y, y'),$$

$$y(x_0) = y_0,$$
$$y(x_1) = y_1,$$

zagadnieniem początkowym postaci

$$y_a'' = f(x, y_a, y_a'),$$

$$y_a(x_0) = y_0,$$
$$y_a'(x_0) = a,$$

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-Kutta.

Kod użyty do wygenerowanie wyników tego zadania przedstawiono poniżej.

```
from math import sin, cos, pi


# y'' = f(x, y, y')
# y(x_0) = y_0
# y(x_1) = y_1
# replacing above problem with
```

```python
# y_a'' = f(x, y_a, y_a')
# y_a(x_0) = y_0
# y_a'(x_0) = a
# and looking for 'a' that implies y_a(x_1) = y_1


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


# y'' = f(x, y, y')
# y(x_0) = y_0
# y(x_1) = y_1
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) # numbers of iterations needed to
        reach x_1 from x_0 with step h

    # looking for 'a' using bisection method - root of F(a) = y_a(x_1) - y_1
    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:
        # F(a) * F(a_0) > 0
        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

    # now we have
    # y_a'' = f(x, y_a, y_a')
    # y_a(x_0) = y_0
    # y_a'(x_0) = a
    return Runge_Kutta_for_hit(final_iterations, h, x_0, y_0, a, func)


if __name__ == '__main__':
    for x_val in [0.5, 1, 2]:
        for n in [100, 1000, 10000, 100000, 1000000]:
            x, y = hit_method(n, -100, 100, 0, 1, pi / 2, pi / 2 - 1, f, x_val
                / n, 1e-3)
            print(f"x={x:.2f}, error={abs(y - exact_solution(x))}")
        print()
```

Otrzymano następujące wyniki.

x	iterations	absolute error
0.5	100	$3.60 * 10^{-4}$
0.5	1000	$3.61 * 10^{-5}$
0.5	10000	$3.61 * 10^{-6}$
0.5	100000	$3.61 * 10^{-7}$
0.5	1000000	$3.61 * 10^{-8}$
1	100	$8.69 * 10^{-5}$
1	1000	$1.06 * 10^{-5}$
1	10000	$1.08 * 10^{-6}$
1	100000	$1.08 * 10^{-7}$
1	1000000	$1.08 * 10^{-8}$
2	100	$2.43 * 10^{-5}$
2	1000	$2.57 * 10^{-4}$
2	10000	$2.55 * 10^{-5}$
2	100000	$2.55 * 10^{-6}$
2	1000000	$2.55 * 10^{-7}$

Wnioski: dokładność metody strzałów jest zaskakująco duża jak na złożoność zagadnienia, do którego rozwiązywania jest przeznaczona - zagadnienie brzegowe jest dużo bardziej złożonym problemem niż zagadnienie początkowe. Mimo tego jesteśmy w stanie uzyskać dokładność rzędu 10^{-7} dla argumentów bliskich x_0 oraz x_1.

3 Bibliografia

https://home.agh.edu.pl/~funika/mownit/lab10
https://pl.wikipedia.org/wiki/Metoda_Eulera

https://pl.wikipedia.org/wiki/Algorytm_Rungego-Kutty
https://pl.wikipedia.org/wiki/Metoda_strza%C5%82%C3%B3w
http://www.if.pw.edu.pl/~agatka/numeryczne/wyklad_09.pdf