Laboratorium X

Rozwiązywanie równań różniczkowych zwyczajnych

Dominik Marek

23 kwietnia 2024

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1. Zadania

Zadanie 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

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

Zadanie 1.2:

Dane jest zagadnienie brzegowe:

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

Znaleźć rozwiązanie metodą strzałów.

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

2.Rozwiązania

2.1

Metoda Runnego-Kutty opiera się na poniższym wzorze rekurencyjnym:

$$y_{n+1} = y_n + \Delta y_n$$

$$\Delta y_n = \frac{(k_1 + 2k_2 + 2k_3 + k)}{6}$$

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = hf\left(x_n + \frac{b}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

Metoda Eulera to szczególny przypadek metody Rungego-Kutty, gdzie:

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

Precyzja	Liczba iteracji	Błąd bezwzględny metody Rungego-Kutty	Błąd bezwzględny metody Eulera
0.01	100	1.9702350861905416e-11	0.0007407326238348944
0.001	1000	1.9984014443252818e-15	7.339644285969671e-05
0.0001	10000	2.248201624865942e-14	7.332909375712404e-06
0.00001	100000	6.196709811945311e-13	7.332230059775569e-07
0.000001	1000000	3.0789815141929466e-12	7.332477119925684e-08
0.02	100	4.044492518673337e-10	0.004450216365194715
0.002	1000	4.035660694512444e-14	0.0004489107292322547
0.0002	10000	2.3370194668359545e-14	4.493035878394558e-05
0.00002	100000	2.0539125955565396e-15	4.493429133223259e-06
0.000002	1000000	2.5390245461665018e-12	4.493493704904594e-07

Powyższe wyniki zostały otrzymane po wykonaniu poniższego programu napisanego w języku Python:

```
from numpy import  e, sin, cos
from typing import Callable


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

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


def euler_method(x0:float, y0:float, h:float, f:Callable,
interation_num:int)-> tuple[float, float]:
    x_curr = x0
    y_curr = y0
    for _ in range(interation_num):
        k1 = h * f(x_curr, y_curr)
        x_curr += h
```

```
        y_curr += k1
    return x_curr, y_curr


def runge_kutta_method(x0:float, y0:float, h: float,
f:Callable,interation_num: int):
    x_curr = x0
    y_curr =  y0
    for _ in range(interation_num):

        k1 = h*f(x_curr, y_curr)
        k2 = h*f(x_curr+h/2, y_curr+k1/2)
        k3 = h*f(x_curr+h/2, y_curr+k2/2)
        k4 = h*f(x_curr+h, y_curr+k3)


        x_curr += h
        y_curr += (k1+ 2*k2 + 2*k3 + k4)/6

    return x_curr, y_curr




def main() -> None:
    interation_number = [100, 1000, 10000, 100000, 1000000]


    for i  in [1, 2]:
        for n in interation_number:
            x, y = runge_kutta_method(0, 0, i/n, fun, n)
            exact_res = exact_result(x)
            print(f'Runge Kutta method:')
            print(f'{i/n=}')
            print(f'{n=}')
            print(f'{x=}')
            print(f'{y=}')
            print(f'{exact_res=}')
            print(f'{abs(exact_res-y)=}')
            x1, y1 = euler_method(0, 0, i/n, fun, n)
            exact_res1 = exact_result(x1)
            print(f'Euler method:')
            print(f'{i/n=}')
            print(f'{n=}')
            print(f'{y1=}')
            print(f'{exact_res=}')
            print(f'{abs(exact_res1 - y1)=}')



if __name__ == '__main__':
    main()
```

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 x0 = 0).

2.2

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) = $y_a\ (x_0) - y_1$. Parametru a można poszukiwać np. metodą bisekcji w połączeniu z metodą Rungego-Kutta.

Precyzja	Liczba iteracji	Błąd bezwzględny metody Rungego-Kutty
0.005	100	0.00036005705082098327
0.0005	1000	3.608712073366327e-05
0.00005	10000	3.6095300024463484e-06
0.000005	100000	3.6096151900810725e-07
0.0000005	1000000	3.609484888755787e-08
0.01	100	8.697235448218432e-05
0.001	1000	1.057578239682666e-05
0.0001	10000	1.076485546702699e-06
0.00001	100000	1.0783849135886925e-07
0.000001	1000000	1.0783245851797574e-08
0.02	100	2.4327633690202077e-05
0.002	1000	0.0002574456118321633
0.0002	10000	2.553221356016433e-05
0.00002	100000	2.5510988537202905e-06
0.000002	1000000	2.5507632561705407e-07

Powyższe zestawienie uzyskano wykonując poniższy skrypt napisany w języku Python:

```
from numpy import sin, cos, pi

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


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


def runge_kutta_for_hit(num_iteration, h, x0, y0, a, func):
    x_curr = x0
    y_curr = y0
    for _ in range(num_iteration):
        k1 = h * func(x_curr, y_curr, a)
        k2 = h * func(x_curr + h / 2, y_curr + k1 / 2, a)
        k3 = h * func(x_curr + h / 2, y_curr + k2 / 2, a)
        k4 = h * func(x_curr + h, y_curr + k3, a)

        delta_a = (k1 + 2 * k2 + 2 * k3 + k4) / 6

        k1 = h * a
        k2 = h * (a +  h * func(x_curr + h / 2, y_curr + k1 / 2, a))
        k3 = h * (a + h*func(x_curr + h / 2, y_curr + k2 / 2, a))
        k4 = h * (a + h*func(x_curr + h, y_curr + k3, a))

        delta_y = (k1 + 2 * k2 + 2 * k3 + k4) / 6

        x_curr += h
        y_curr += delta_y
        a += delta_a


    return x_curr, y_curr
# y'' = f(x, y, y')
# y(x0) = y0
# y(x1) = y1


def hit_method(final_iterations, a0, a1, x0, y0, x1,y1, func, h, epsilon):
    # Number of iterations to reach x1 from x0 with step h
    bisect_iterations = int((x1-x0)/h)
    a = (a0+a1)/2
    y =  runge_kutta_for_hit(bisect_iterations, h, x0, y0, a, func)[1]
    i = 0

    while abs(y-y1) > epsilon:
        if (y-y1)*(runge_kutta_for_hit(bisect_iterations, h, x0, y0, a0,
func)[1] -y1) > 0:
            a0 = a
        else:
            a1 = a

        a = (a0 + a1)/2

        y = runge_kutta_for_hit(bisect_iterations, h, x0, y0, a, func)[1]
        i += 1
    # now we have
    # ya'' = f(x, ya, ya')
```

```
    # ya(x0) =  y0
    # ya' =  a

    return runge_kutta_for_hit(final_iterations, h, x0, y0, a, func)



def main()->None:
    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, fun,
x_val/n, 1e-3)
            print(f'{x_val/n =}')
            print(f'{abs(y-exact_result(x))=}')


if __name__ == '__main__':
    main()
```

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.

3. Bibliografia

- https://home.agh.edu.pl/~funika/mownit/lab10 https://pl.wikipedia.org/wiki/Metoda_Eulera 5
- 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