Kacper Korta,

30.05.2023

Laboratorium 11

Rozwiązywanie równań różniczkowych zwyczajnych

1. Treść zadań

Zadanie 1:

Dane jest równanie różniczkowe (zagadnienie początkowe):

y’ + y cosx = sinx cosx y(0) = 0

Znaleźć rozwiązanie metodą Rungego-Kutty i metodą Eulera.

Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym y(x) = e-sinx + sinx – 1.

Zadanie 2:

Dane jest zagadnienie brzegowe:

y’’ + y = x y(0) = 1 y(0.5p) = 0.5p - 1

Znaleźć rozwiązanie metodą strzałów.

Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym y(x) = cosx – sinx + x .

2. Rozwiązanie

Zadanie 1.

Korzystając z metody Rungego-Kutty 4. Rzędu otrzymujemy wzory:

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

Gdzie

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

Po zaimplementowaniu powyższych wzorów w pythonie otrzymujemy taki kod:

```
from math import sin, cos, e

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

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


def RungeKutta(n, h, x0, y0):
    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
```

Do porównania wyników metody przyjąłem n = 10, 100, 1000, 10000, 100000.

Po uruchumieniu programu otrzymaliśmy:

```
n =  10 RungeKutta result:  0.2725471473929363 , Δy:  2.1193944743647108e-07
n =  100 RungeKutta result:  0.2725469354731914 , Δy:  1.9702350861905416e-11
n =  1000 RungeKutta result:  0.27254693545349107 , Δy:  1.9984014443252818e-15
n =  10000 RungeKutta result:  0.27254693545348246 , Δy:  2.248201624865942e-14
n =  100000 RungeKutta result:  0.27254693545351943 , Δy:  6.196709811945311e-13
```

Metoda Eulera napisana w Pythonie:

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

otrzymane wyniki

```
n =  10 Euler result:  0.26442725830740726 , Δy:  0.008119677146081583
n =  100 Euler result:  0.2718062028296542 , Δy:  0.0007407326238348944
n =  1000 Euler result:  0.2724735390106294 , Δy:  7.339644285969671e-05
n =  10000 Euler result:  0.27253960254408427 , Δy:  7.332909375712404e-06
n =  100000 Euler result:  0.2725462022298938 , Δy:  7.332230059775569e-07
```

Wnioski: Na podstawie wyników metody Rungego-Kutty i metody Eulera można wyciągnąć następujące wnioski:

Metoda Rungego-Kutty jest bardziej dokładna niż metoda Eulera. Porównując wyniki dla różnych wartości n (liczba kroków) można zauważyć, że wyniki uzyskane za pomocą metody Rungego-Kutty są

znacznie bliższe oczekiwanemu wynikowi (0.272546935453). Błąd (△y) jest również znacznie mniejszy dla metody Rungego-Kutty.

Im większa liczba kroków (n), tym dokładniejsze są wyniki. Dla obu metod można zauważyć, że im większa liczba kroków, tym bardziej wyniki zbliżają się do oczekiwanego wyniku. Dla metody Rungego-Kutty błąd (△y) maleje bardzo szybko wraz ze zwiększaniem liczby kroków. Dla metody Eulera błąd maleje również, ale nie tak szybko jak w przypadku metody Rungego-Kutty.

Metoda Eulera jest prostsza do zaimplementowania niż metoda Rungego-Kutty. Metoda Eulera polega na prostym iteracyjnym obliczaniu kolejnych wartości na podstawie poprzednich, co jest stosunkowo proste do zrozumienia i zaimplementowania. Z drugiej strony, metoda Rungego-Kutty wymaga bardziej złożonych obliczeń, ale daje dokładniejsze wyniki.

Zadanie 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

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

Kod napisany w Pythonie

```
from math import sin, cos, pi

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

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


def RungeKutta(n, h, x0, y0, a, func):
    x = x0
    y = y0
    for _ in range(n):
        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 final_sol(n, a0, a1, x0, x1, y0, y1, func, h, epsilon):
    bisect_iter = int((x1 - x0) / h)
    a = (a0 + a1) / 2
    y = RungeKutta(bisect_iter, h, x0, y0, a, func)[1]
    i = 0
    while abs(y - y1) > epsilon:
        if (y - y1) * (RungeKutta(bisect_iter, h, x0, y0, a0,
func)[1] - y1) > 0:
            a0 = a
        else:
            a1 = a
        a = (a0 + a1) / 2
        y = RungeKutta(bisect_iter, h, x0, y0, a , func)[1]
        i += 1

    return  RungeKutta(n, h, x0, y0, a , func)
```

Uruchamiając program otrzymaliśmy następujące wyniki:

```
X: 0.5 , Iterations:  10 , Δy =  0.00645433832982012
X: 0.5 , Iterations:  100 , Δy =  0.00036005705082098327
X: 0.5 , Iterations:  1000 , Δy =  3.608712073366327e-05
X: 0.5 , Iterations:  10000 , Δy =  3.6095300024463484e-06
X: 0.5 , Iterations:  100000 , Δy =  3.6096151900810725e-07
X: 1 , Iterations:  10 , Δy =  0.01209992711708463
X: 1 , Iterations:  100 , Δy =  8.697235448218432e-05
X: 1 , Iterations:  1000 , Δy =  1.057578239682666e-05
X: 1 , Iterations:  10000 , Δy =  1.076485546702699e-06
X: 1 , Iterations:  100000 , Δy =  1.0783849135886925e-07
X: 2 , Iterations:  10 , Δy =  0.0077572097515643534
X: 2 , Iterations:  100 , Δy =  2.4327633690202077e-05
X: 2 , Iterations:  1000 , Δy =  0.0002574456118321633
X: 2 , Iterations:  10000 , Δy =  2.553221356016433e-05
X: 2 , Iterations:  100000 , Δy =  2.5510988537202905e-06
```

Wnioski:
Na podstawie podanych wyników można wyciągnąć następujące wnioski dotyczące metody strzałów:

Wartość X ma wpływ na wartość ⧋y, przy czym dla wartości X równych 0.5 i 1, ⧋y maleje wraz z zwiększaniem liczby iteracji, a dla X równego 2, ⧋y wzrasta wraz z zwiększaniem liczby iteracji.

Dla wartości X równych 0.5, 1 i 2, im większa liczba iteracji, tym mniejsza wartość ⧋y. Oznacza to, że metoda strzałów zwiększa dokładność rozwiązania wraz ze zwiększaniem liczby iteracji.

Dla każdej wartości X, dla dostatecznie dużej liczby iteracji, ⧋y osiąga bardzo małe wartości, zbliżone do zera. Oznacza to, że metoda strzałów jest skuteczną metodą numeryczną do rozwiązywania równań, zapewniając bardzo dokładne wyniki dla odpowiednio dużego zwiększenia liczby iteracji.

Dla danej liczby iteracji, wartość ⧋y jest najmniejsza dla X równego 1. Oznacza to, że metoda strzałów jest najbardziej skuteczna dla tej wartości X w kontekście minimalizowania błędu numerycznego.

3. Bibliografia

Wykład
https://en.wikipedia.org/wiki/Shooting_method
https://en.wikipedia.org/wiki/Euler_method
https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods