Zadanie 4 — Całkowanie numeryczne

1 Polecenie

Celem zadania czwartego jest stworzenie programu implementującego dwie metody całkowania numerycznego:

1. złożoną kwadraturę Newtona-Cotesa opartą na trzech węzłach (wzór Simpsona)
2. wariant kwadratury Gaussa: całkowanie na przedziale $[a, b)$ (wielomiany Legendre'a) całek postaci $\int_a^b f(x)dx$

☒ Kwadratury złożone Newtona-Cotesa obliczane są z dokładnością podaną przez użytkownika.
☒ Odbywa się to w sposób iteracyjny.
☒ W każdej iteracji ilość podprzedziałów na które podzielony jest przedział całkowania jest zwiększana, a otrzymany wynik całkowania porównywany jest z wynikiem z poprzedniej iteracji.
☒ Jeśli wynik zmienił się o mniej niż dokładność podana przez użytkownika, oznacza to że dokładność całki na podanym przedziale została obliczona z zadaną dokładnością.

Przy obliczaniu całki na przedziale $[0, +\infty)$ stosuje się następujące podejście:

1. obliczenie całki na przedziale $[0, a)$, gdzie $a > 0$,
2. obliczanie całki na przedziale $[a, a + \delta)$, gdzie $\delta > 0$.
   - Jeśli wartość całki na tym przedziale jest większa od zakładanej dokładności, to otrzymany wynik dodawany jest do wcześniejszego wyniku,
     - przyjmuje się $a := a + \delta$, po czym operacja jest powtarzana.
   - Jeśli nie uznaje się, że policzyło się granicę dążącą do $+\infty$.
   - Analogicznie postępuje się obliczając wartość całki na przedziale $(-\infty, 0]$. W przypadku obliczania granicy dążącej do ± 1 postępuje się w sposób podobny, przy czym najpierw liczy się wartość całki na przedziale $[0, 0.5)$, potem dolicza się wartość na przedziale $[0.5, 0.5 + 0.5 \cdot 0.5)$, następnie na przedziale $[0.75, 0.75 + 0.25 \cdot 0.5)$ i tak dalej.

☒ Kwadratury Gaussa obliczane są dla 2, 3, 4 i 5 węzłów.

☒ konieczne jest przeskalowanie funkcji i granic całkowania na przedział $[-1, 1)$.

☒ W sprawozdaniu należy porównać wyniki uzyskane za pomocą obu metod całkowania.

☒ Należy pamiętać, że funkcje całkowane są postaci $w(x) \cdot f(x)$, gdzie $w(x)$ to funkcja wagowa, przy czym w kwadraturach Gaussa funkcja wagowa jest od razu uwzględniona w metodzie.

☒ Przy obliczaniu kwadratur Newtona-Cotesa trzeba więc dodać funkcję wagową. # Program ## Importy i stałe Stanardowo wykorzystaliśmy biblioteki `math` i `numpy`, oraz stworzoną przez nas wcześniej funkcję `hornerThis`:

[1]:
```
import math; from math import sin, cos
import numpy as np

INTERACTIVE = False

def hornerThis(x, coefs):
    rval = 0
    for i in coefs: rval = rval*x + coefs[i]
    return rval
```

do wyboru mamy następujące funkcje:

1. $x + 1$
2. $\cos(x) - \sin(x)$
3. $-x^2 + x - 1$
4. $|x - 5|$
5. $\frac{\cos(x)}{x}$

[2]:
```
def fun(func_i, x):
    match func_i:
        case 1: return x + 1
        case 2: return cos(x) - sin(x)
        case 3: return hornerThis(x, (-3, 2, -1))
        case 4: return math.fabs(x - 5)
        case 5: return cos(x) /x
```

[3]:
```
def simpThis(func_i, a, b, eps):
    subinterval = 1
    length_all = b - a
    result = 0
    result_previous = eps +1

    while abs(result - result_previous) > eps:
        subinterval *= 2
        length_sub = length_all / subinterval
        result_previous = result
        result = 0

        x = np.linspace(a, b, subinterval + 1)

        for i in range(0, math.floor(subinterval/2)):
            result +=   fun(func_i, x[2*i])
            result += 4*fun(func_i, x[2*i+1])
            result +=   fun(func_i, x[2*i+2])

        result *= length_sub / 3
    return result
```

[4]:
```
def glThis(func_i, l_edg, r_edg, nodes): # Gauss Legendre
    coefficients = {
        1: (),
        2: ((-0.5773502691896257, 1), (0.5773502691896257, 1)),
        3: ((-0.7745966692414834, 0.5555555555555556), (0, 0.8888888888888888),
            (0.7745966692414834, 0.5555555555555556)),
        4: ((-0.8611363115940526, 0.3478548451374538), (-0.33998104358485626, 0.
↪6521451548625461),
            (0.33998104358485626, 0.6521451548625461), (0.8611363115940526, 0.
↪3478548451374538)),
        5: ((-0.906179845938664, 0.23692688505618908), (-0.5384693101056831, 0.
↪47862867049936647),
            (0, 0.5688888888888889),
            (0.5384693101056831, 0.47862867049936647), (0.906179845938664, 0.
↪23692688505618908))
    }

    integral = 0
    for i in range(nodes):
        xi, wi = coefficients[nodes][i]
        xi_mapped = ((r_edg - l_edg) * xi + (l_edg + r_edg)) / 2
        integral += wi * fun(func_i, xi_mapped)

    integral *= (r_edg - l_edg) / 2

    return integral
```

[5]:
```
def last_function(func_i, l_edg, r_edg, eps=0.01, node_c=6, VERBOSE=True,␣
↪TALKBACK=True):
    if TALKBACK:
        print(f"Wyniki dla funkcji {func_i} na przedziale od {l_edg} do {r_edg}␣
↪z dokładnością {eps}:")
    simp = simpThis(func_i, l_edg, r_edg, eps)
    if VERBOSE: print(f"Simpson: \t\t {simp}")
    gl = []
    for i in range(2, node_c):
        gl_part = glThis(func_i, l_edg, r_edg, i)
        gl.append(gl_part)
        if VERBOSE: print(f"Gauss-Legrendge ({i}): \t {gl_part}")
        else: return simp, gl
```

2 Badania

[6]:
```
last_function(1,0,5)
```

```
Wyniki dla funkcji 1 na przedziale od 0 do 5 z dokładnością 0.01:
Simpson:                 17.5
```

```
Gauss-Legrendge (2):     17.5
Gauss-Legrendge (3):     17.5
Gauss-Legrendge (4):     17.499999999999996
Gauss-Legrendge (5):     17.5
```

[7]:
```
last_function(2,0,2*math.pi)
```

```
Wyniki dla funkcji 2 na przedziale od 0 do 6.283185307179586 z
dokładnością 0.01:
Simpson:                 5.813114163347719e-17
Gauss-Legrendge (2):     1.5118507150637253
Gauss-Legrendge (3):     -0.14108390249249086
Gauss-Legrendge (4):     0.006709407061961377
Gauss-Legrendge (5):     -0.00019354294514008878
```

[8]:
```
last_function(3,0,5)
```

```
Wyniki dla funkcji 3 na przedziale od 0 do 5 z dokładnością 0.01:
Simpson:                 -142.5
Gauss-Legrendge (2):     -142.50000000000003
Gauss-Legrendge (3):     -142.5
Gauss-Legrendge (4):     -142.5
Gauss-Legrendge (5):     -142.5
```

[9]:
```
last_function(4,0,10)
```

```
Wyniki dla funkcji 4 na przedziale od 0 do 10 z dokładnością 0.01:
Simpson:                 25.0
Gauss-Legrendge (2):     28.86751345948129
Gauss-Legrendge (3):     21.516574145596756
Gauss-Legrendge (4):     26.063371431538176
Gauss-Legrendge (5):     23.621260909976954
```

[10]:
```
last_function(5, 2*math.pi, 4*math.pi)
```

```
Wyniki dla funkcji 5 na przedziale od 6.283185307179586 do 12.566370614359172 z
dokładnością 0.01:
Simpson:                 0.016321255425214027
Gauss-Legrendge (2):     0.16658204851343658
Gauss-Legrendge (3):     0.005125306540896187
Gauss-Legrendge (4):     0.01685978138601474
Gauss-Legrendge (5):     0.016434628819204163
```

3 Część interaktywna

Specyfikacja zadania nakazuje więc my dowozimy.

[11]:
```
def printChoices():
    print("Dostępne funkcje:")
```

```
    print("1. x + 1")
    print("2. cos(x) - sin(x)")
    print("3. -x^2 + x - 1")
    print("4. |x - 5|")
    print("5. cos(x) / x")
def queryDetails():
    func_i = int(input("Wybierz numer funkcji"))
    l_edg = float(input("Podaj początek przedziału: "))
    r_edg = float(input("Podaj koniec przedziału: "))
    acc =   float(input("Podaj dokładność: "))
    return func_i, l_edg, r_edg, acc

if INTERACTIVE:
    printChoices()
    func_i, l_edg, r_edg, acc = queryDetails
    last_function(func_i, l_edg, r_edg, acc)
```

4 Wnioski

Metoda simpsona przynosi dokładniejsze wyniki, podczas gdy metoda Gaussa-Legandre'a zwiększa swoją dokładność wraz ze zwiększeniem liczby węzłów.

Ponadto, można zauważyć niedoskonałości systemu zapisu `float` (IEEE 754).