Laboratorium 5 - Całkowanie numeryczne

Piotr Karamon

16.04.2024r.

Treści zadań

Zadania

1. Obliczyć $I = \int_0^1 \frac{1}{(1+x)} dx$ wg wzoru prostokątów, trapezów i wzoru Simpsona (zwykłego i złożonego $n = 3, 5$). Porównać wyniki i błędy.
2. Obliczyć całkę $I = \int_{-1}^1 \frac{1}{(1+x^2)} dx$ korzystając z wielomianów ortogonalnych (np. Czebyszewa) dla $n = 8$.

Zadania domowe

1. Obliczyć całkę $\int_0^1 \frac{1}{(1+x^2)} dx$ korzystając ze wzoru prostokątów dla $h = 0.1$ oraz metody całkowania adaptacyjnego.
2. Metodą Gaussa obliczyć następującą całkę $\int_0^1 \frac{1}{(x+3)} dx$ dla $n = 4$. Oszacować resztę kwadratury.

Zadanie 1

Chcemy obliczyć całkę numerycznie.

$$I = \int_0^1 \frac{1}{(1+x)} dx$$

W celu wyznaczenia błędu obliczmy dokładną wartość całki

$$I = \int_0^1 \frac{1}{(1+x)} dx = \ln(1+x)\Big|_1^0 = \ln 2$$

Na początku stworzymy funkcje testującą różne wersje liczenia tej całki numerycznie.

```
import numpy as np

def f(x):
    return 1/(1+x)

def tester(ns, approximate):
    for n in ns:
        approx = approximate(f, 0, 1, n)
        abs_error = np.abs(np.log(2) - approx)
        rel_error = np.round(abs_error / np.log(2) * 100, 3)
        print(f'n: {n},\n\tapprox =    {approx},\n\tabs err =   {abs_error},\n\trel_error =
        ↪  {rel_error}%')
```

Metoda prostokątów

Polega na podzieleniu przedziału całkowania na n mniejszych przedziałów i zsumowaniu pól prostokątów których wysokością jest wartość funkcji w środowym punkcie małego przedziału a szerokością $\frac{b-a}{n}$.

$$S(f) = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i + \frac{1}{2} \cdot \frac{b-a}{n})$$

$$x_i = a + \frac{b-a}{n} \cdot i$$

```
def rectangle(f, a, b, n):
    xs = np.arange(a, b, (b-a)/n) + (b-a)/(2*n)
    integral = (b-a)/n * np.sum(f(xs))
    return integral

tester([1,3,5], rectangle)
```

```
n: 1,
        approx =    0.6666666666666666,
        abs err =   0.026480513893278657,
        rel_error = 3.82%
n: 3,
        approx =    0.6897546897546897,
        abs err =   0.0033924908052561,
        rel_error = 0.489%
n: 5,
        approx =    0.6919078857159353,
        abs err =   0.001239294844009975,
        rel_error = 0.179%
```

Metoda trapezów

Metoda trapezów polega na przybliżeniu obszaru ograniczonego wykresem funkcji przez trapezy prostokątne o wysokości równej długości kroku całkowania i podstawach o długościach odpowiadających wartościom funkcji w punktach węzłowych na brzegu przedziału.

$$S(f) = \frac{1}{2} \cdot \frac{b-a}{n} \sum_{i=0}^{n-1} [f(x_i) + f(x_{i+1})]$$

```
def trapezoid(f, a, b, n):
    dx = (b-a)/n
    xs = np.linspace(a, b, n+1)
    integral = 0
    for i in range(1, n+1):
        integral += dx*(f(xs[i]) + f(xs[i-1])) /2
    return integral

tester([1,3,5], trapezoid)
```

```
n: 1,
        approx =    0.75,
        abs err =   0.056852819440054714,
        rel_error = 8.202%
n: 3,
        approx =    0.7,
        abs err =   0.006852819440054669,
        rel_error = 0.989%
n: 5,
        approx =    0.6956349206349206,
        abs err =   0.00248774007497532,
        rel_error = 0.359%
```

Metoda Simpsona

Metoda opiera się na przybliżaniu funkcji całkowanej przez interpolację wielomianem drugiego stopnia.

Znając wartości y_0, y_1, y_2 funkcji $f(x)$ w 3 punktach x_0, x_1, x_2 (przy czym $x_2 - x_1 = x_1 - x_0 = h$), przybliża się funkcję wielomianem Lagrange'a i całkując w przedziale $[x_0, x_2]$, otrzymuje przybliżoną wartość całki:

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}(y_0 + 4y_1 + y_2)$$

Zatem

$$S(f) = \frac{b-a}{6} \sum_{i=1}^{n/2} [f(x_{2i-2}) + 4f(x_{2x-1}) + f(x_{2i})]$$

```python
def simpson_rule(f, a, b, n):
    if n % 2 == 1: # n musi być parzyste
        n += 1
    dx = (b-a)/n
    xs = np.linspace(a, b, n+1)
    integral = 0
    for i in range(1, n //2 +1):
        integral += dx/3*( f(xs[2*i-2]) + 4*f(xs[2*i-1]) + f(xs[2*i]) )
    return integral

tester([3, 5], simpson_rule)
```

```
n: 3,
        approx =    0.6932539682539682,
        abs err =   0.0001067876940229473,
        rel_error = 0.015%
n: 5,
        approx =    0.6931697931697931,
        abs err =   2.2612609847816323e-05,
        rel_error = 0.003%
```

Wnioski

Metoda prostokątów i trapezów poprawia dokładność przy większym n podczas numerycznego całkowania funkcji $f(x) = \frac{1}{1+x}$. Pomiędzy metodą prostokątów a trapezów nie widać dramatycznej różnicy w dokładności. Zdecydowanie najlepiej wypadała metoda Simpsona która jest jednocześnie najbardziej złożoną metodą przedstawioną w tym zadaniu. Błąd metody Simpsona w tym przypadku wyszedł około 100 razy mniejszy od pozostałych metod.

Zadanie 2

Zaczynamy od policzenia całki analitycznie w celu weryfikacji użytej metody.

$$I = \int_{-1}^{1} \frac{1}{(1+x^2)} dx = \arctan(x)\Big|_{1}^{-1} = \arctan(1) - \arctan(-1) = \frac{\pi}{2}$$

Z racji na brak czynnika $\frac{1}{\sqrt{1-x^2}}$ lub $\sqrt{1-x^2}$ w naszej funkcji nie skorzystamy z wielomianów Czebyszewa. Wykorzystamy wielomiany Legendre'a.

Wagi mają postać

$$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2}$$

Obliczanie węzłów jest już o wiele bardziej złożone, zatem skorzystamy z biblioteki `numpy` i jej podmodułu `polynmial.legendre`.

i	x_i	w_i
1	-0.9602898564975362	0.10122853629037669
2	-0.7966664774136267	0.22238103445337434
3	-0.525532409916329	0.31370664587788705
4	-0.18343464249564978	0.36268378337836177
5	0.18343464249564978	0.36268378337836177
6	0.525532409916329	0.31370664587788705
7	0.7966664774136267	0.22238103445337434
8	0.9602898564975362	0.10122853629037669

Przybliżenie naszej całki ma następujący wzór:

$$S(f) = \sum_{i=1}^{8} w_i f(x_i)$$

```
from numpy.polynomial.legendre import leggauss
import numpy as np

def f(x):
    return 1 / (1 + x**2)

n = 8

nodes, weights = leggauss(n)
approx = sum(weights * f(nodes))

abs_error = np.abs(np.pi/2 - approx)
rel_error = np.round(abs_error / np.pi/2 * 100, 7)
print(f'n: {n},\n\tapprox =    {approx},\n\tabs err =   {abs_error},\n\trel_error = {rel_error}%')
```

```
n: 8,
        approx =    1.570794412546062,
        abs err =   1.9142488345558206e-06,
        rel_error = 3.05e-05%
```

Wnioski Kwadratura Gaussa-Legendre'a daje bardzo imponujące wyniki. Wzór rekurencyjny dla wielomianów Legendre'a połączony z wyżej przedstawionym wzorem na wagi pozwala łatwo je obliczyć. Problemem w tej metodzie jest znalezienie węzłów, dla $n = 8$ metody analitycznie są niepraktyczne. Trzeba je przybliżać np. metodą Newtona-Raphsona. Jednakże, warto zwrócić uwagę na to, że te obliczenia wykonamy tak na prawdę raz. Potem liczenie całek jest trywialne oraz szybkie.

Zadanie domowe 1

Zaczynamy od obliczenia całki analitycznie

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)\Big|_0^1 = \frac{\pi}{4}$$

Użyta technika całkowania to wzór prostokątów oraz całkowanie adaptacyjne.

Pierwsze przybliżenie całki jest obliczane dla całego przedziału $[a, b]$(w naszym przypadku $[0, 1]$). Następnie ten przedział jest dzielony na połówki. Wzór prostokątów jest teraz stosowany osobno dla lewej połówki (od a do $c = \frac{(a+b)}{2}$) i dla prawej (od c do b). Otrzymujemy przybliżone wartości dwóch części obliczanej całki – $S1$ i $S2$. Jeżeli $|S1 + S2 - S| < \Delta$ to uznajemy, że otrzymany wynik jest dostatecznie dokładny i kończymy algorytm. W przeciwnym przypadku stajemy przed dwoma zadaniami - obliczyć całkę na dwóch połówkach (lewej i prawej), każdej z błędem nie większym niż $\Delta/2$.

W programie znajduje się również kontrola poziomu zagnieżdżenia, w przypadku jego przekroczenia zwracamy wartość nan.

```
import math

def adaptive_integral(f, a, b, delta=1e-5, max_level=10):
    initial_estimate = midpoint_rectangular_quadrature(f, a, b)
    return recursive_integral(f, a, b, initial_estimate, delta, 0, max_level)


def midpoint_rectangular_quadrature(f, a, b):
    midpoint = (a + b) / 2
    return f(midpoint) * (b - a)

def recursive_integral(f, a, b, S, delta, level, max_level=10):
    if level == max_level:
        return float('nan')

    S1 = midpoint_rectangular_quadrature(f, a, (a + b) / 2)
    S2 = midpoint_rectangular_quadrature(f, (a + b) / 2, b)

    if abs(S1 + S2 - S) <= delta:
        return S1 + S2
    else:
        left = recursive_integral(f, a, (a + b) / 2, S1, delta / 2, level + 1, max_level)
        right = recursive_integral(f, (a + b) / 2, b, S2, delta / 2, level + 1, max_level)
        return left + right
```

Teraz przetestujemy napisaną procedurę dla zadanej funkcji:

```
import numpy as np

def f(x):
    return 1/(1+x**2)

for delta in [0.1, 0.05, 1e-3, 1e-5, 1e-10]:
    approx = adaptive_integral(f, 0, 1, delta, 30)

    abs_error = np.abs(np.pi/4 - approx)
    rel_error = np.round(abs_error / np.pi/4 * 100, 7)
    print(f'delta: {delta},\n\tapprox =    {approx},\n\tabs err =   {abs_error},\n\trel_error =
    ↪  {rel_error}%')
```

```
delta: 0.1,
        approx =    0.7905882352941176,
        abs err =   0.005190071896669313,
        rel_error = 0.0413013%
delta: 0.05,
        approx =    0.7905882352941176,
        abs err =   0.005190071896669313,
        rel_error = 0.0413013%
delta: 0.001,
        approx =    0.7854101544620723,
        abs err =   1.1991064624017334e-05,
        rel_error = 9.54e-05%
delta: 1e-05,
        approx =    0.7853984247972766,
        abs err =   2.613998283385044e-07,
        rel_error = 2.1e-06%
delta: 1e-10,
        approx =    0.7853981634001626,
        abs err =   2.7142732506035827e-12,
        rel_error = 0.0%
```

Wnioski: Jak widzimy całkowanie adaptacyjne pozwala nam dobierać precyzję z jaką chcemy otrzymać wynik. Oczywiście mamy tutaj wybór pomiędzy dwoma sprzecznymi korzyściami. Jeżeli zaznaczymy, że chcemy szybko(mało iteracji) policzyć całkę to musimy poświęcić precyzję. Z kolei policzenie całki bardzo dokładnie wymaga wielu rekurencyjnych wywołań.

Zadanie domowe 2

Liczmy dokładną wartość całki.

$$\int_0^1 \frac{1}{(x+3)}dx = \ln(x+3)\Big|_0^1 = \ln(4)\ln - \ln(3) = \ln\left(\frac{4}{3}\right)$$

Użyjemy kwadratury Gauss-Legendre'a. Najpierw wyliczmy potrzebne nam wielomiany Legendre'a.

```python
from sympy import Rational, symbols, expand, lambdify
x = symbols('x')


def legendre(n):
    match n:
        case 0: return 1
        case 1: return x
        case _:
            return expand(Rational(1, n)*((2*n-1)*x*legendre(n-1) - (n-1)*legendre(n-2)))

for i in range(0, 5):
```

```
    print(f"legendre({i}) = {legendre(i)}")

L = list(map(lambda n: lambdify(x, legendre(n)), range(0, 5)))
```

```
legendre(0) = 1
legendre(1) = x
legendre(2) = 3*x**2/2 - 1/2
legendre(3) = 5*x**3/2 - 3*x/2
legendre(4) = 35*x**4/8 - 15*x**2/4 + 3/8
```

A zatem nasze funkcję to :

$$\begin{aligned}
\phi_0(x) &= 1 \\
\phi_1(x) &= x \\
\phi_2(x) &= \frac{3x^2}{2} - \frac{1}{2} \\
\phi_3(x) &= \frac{5x^3}{2} - \frac{3x}{2} \\
\phi_4(x) &= \frac{35x^4}{8} - \frac{15x^2}{4} + \frac{3}{8}
\end{aligned}$$

W celu policzenia węzłów rozwiążemy równanie

$$\phi_4(x) = \frac{35x^4}{8} - \frac{15x^2}{4} + \frac{3}{8} = 0$$

Zrobimy to za pomocą biblioteki Sympy.

```
from sympy import Eq, solve, latex, N

eq = Eq(L[4], 0)
solutions = sorted(solve(eq, x), key=lambda sol: sol.evalf())
print(latex(solutions))

nodes = [sol.evalf() for sol in solutions]
print(nodes)
```

Węzły są równe:

$$\left[- \sqrt{\frac{2\sqrt{30}}{35} + \frac{3}{7}}, \ - \sqrt{\frac{3}{7} - \frac{2\sqrt{30}}{35}}, \ \sqrt{\frac{3}{7} - \frac{2\sqrt{30}}{35}}, \ \sqrt{\frac{2\sqrt{30}}{35} + \frac{3}{7}}\right]$$

To ich przybliżenia

$$x_1 = -0.861136311594053$$
$$x_2 = -0.339981043584856$$
$$x_3 = 0.339981043584856$$
$$x_4 = 0.861136311594053$$

Aby obliczyć współczynniki(wagi) rozwiązujemy następujące równianie

$$\begin{bmatrix} \phi_0(x_1) & \phi_0(x_2) & \phi_0(x_3) & \phi_0(x_4) \\ \phi_1(x_1) & \phi_1(x_2) & \phi_1(x_3) & \phi_1(x_4) \\ \phi_2(x_1) & \phi_2(x_2) & \phi_2(x_3) & \phi_2(x_4) \\ \phi_3(x_1) & \phi_3(x_2) & \phi_3(x_3) & \phi_3(x_4) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} \int_{-1}^{1} w(x)\phi_0(x)dx \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\int_{-1}^{1} w(x)\phi_0(x)dx = \int_{-1}^{1} 1 \cdot 1dx = 2$$

Jesteśmy gotowi by policzyć układ równań, korzystamy z utworzonej wcześniej tablicy `L` (wielomiany Legendre'a) i tablicy `nodes` (węzły)

```
import numpy as np

A = np.zeros((4,4))
for i in range(4):
    for j in range(4):
        A[i,j] = L[i](nodes[j])
b = np.zeros(4)
b[0] = 2

solution =np.linalg.solve(A, b)

print(solution)
```

```
[0.34785485 0.65214515 0.65214515 0.34785485]
```

Podsumowując:

Węzły (x_i)	**Wagi** (w_i)
-0.861136311594053	0.34785485
-0.339981043584856	0.65214515
0.339981043584856	0.65214515
0.861136311594053	0.34785485

Przedział naszej całki to $[0,1]$ a nie $[-1,1]$ musimy zatem ją przekształcić. Podstawiamy $t = 2x - 1$

$$\int_0^1 \frac{1}{x+3}dx = \int_{-1}^{1} \frac{1}{t+7}dt$$

```python
import numpy as np

xs = np.array([-0.861136311594053,
               -0.339981043584856,
               0.339981043584856,
               0.861136311594053])

ws = np.array([0.34785485,
               0.65214515,
               0.65214515,
               0.34785485])
def f(t):
    return 1/(t + 7)


approx = np.sum(f(xs)*ws)
real = np.log(4/3)
abs_err = abs(approx-real)
rel_err = abs_err/real
print(f'approx  = {approx}')
print(f'real    = {real}')
print(f'abs_err = {abs_err}')
print(f'rel_err = {rel_err}')
```

```
approx  = 0.28768207216869485
real    = 0.28768207245178085
abs_err = 2.8308599908655196e-10
rel_err = 9.840237755308883e-10
```

Wnioski: Błąd tej metody w tym przypadku jest niezwykle mały, pomimo że użyliśmy niewielkiej liczby $n = 4$. Przykład tutaj przedstawiony pokazuje, jak skuteczna i dokładna może być ta metoda numeryczna dla odpowiednio dobranych węzłów i wag. W przypadku gdy całkujemy naszą funkcję po przedziale innym niż $[-1, 1]$ możemy z łatwością, zamienić ją na całkę po tym właśnie przedziale.

Bibliografia

- Marian Bubak, Katarzyna Rycerz: *Metody Obliczeniowe w Nauce i Technice. Kwadratury*
- GaussLegendre
- ChebyshevGauss
- Włodzimierz Funika: Całkowanie