Laboratorium VI

Całkowanie numeryczne

Dominik Marek

16 kwietnia 2024

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1.Zadania

Obliczyć przybliżoną wartość całki:

$$\int_{-\infty}^{\infty} e^{-x^2} cos(x)\, dx$$

a) przy pomocy złożonych kwadratur (prostokątów, trapezów, Simpsona),

b) przy pomocy całkowania adaptacyjnego,

c) przy pomocy kwadratury Gaussa-Hermite'a, obliczając wartości węzłów i wag.

Porównać wydajność dla zadanej dokładności.

2 .Rozwiązania

W poniższych zadaniach będę wykorzystywał fakt, że wartość całki $\int_{-\infty}^{\infty} e^{-x^2}\, cos(x)\, dx$ jest znana i wynosi:

$$S_R(f) = \int_{-\infty}^{\infty} e^{-x^2}\, cos(x)\, dx = \frac{\sqrt{\pi}}{\sqrt[4]{e}} \approx 1.3803884470431430$$

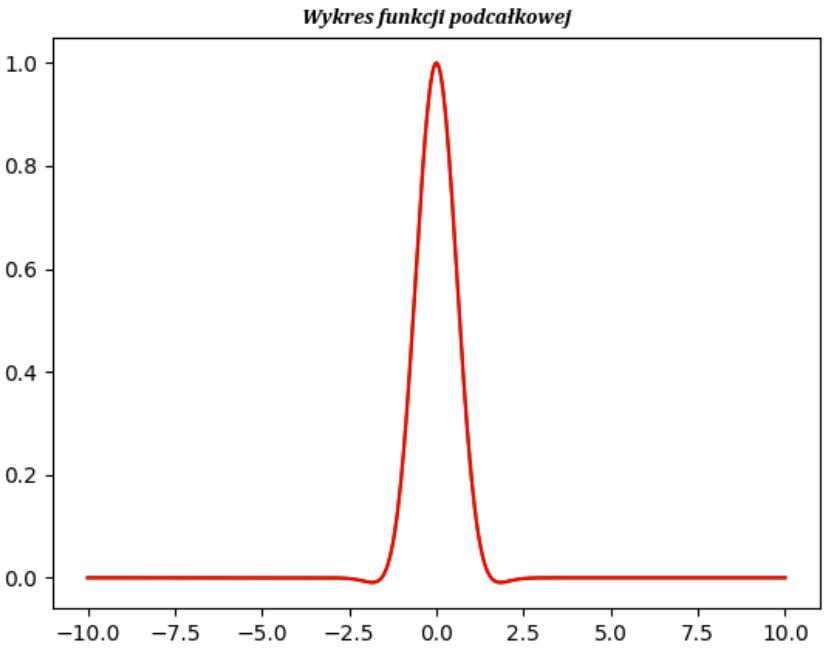
a)

Zadanie rozpoczynam od policzenia granic funkcji w $-\infty$ oraz ∞:

$$\lim_{x\to-\infty} f(x) = \lim_{x\to-\infty} e^{-x^2}\, cos(x) = 0$$

$$\lim_{x\to\infty} f(x) = \lim_{x\to\infty} e^{-x^2}\, cos(x) = 0$$

Obie granice są równe zero, ponieważ e^{-x^2} zarówno w $-\infty$ oraz ∞ zmierza do 0 a $cos(x)$ jest niekreślony, więc korzystając z twierdzenia o iloczynie funkcji niekreślonej i funkcji dążącej do zera możemy stwierdzić, iż wartość obu granic wynosi zero.



Dla poniższych metod przeprowadzałem obliczenia dla zadanej dokładności $\varepsilon = 1 \cdot 10^{-16}$. Dla zadanej dokładności otrzymano następujące wyniki z rozróżnieniem na daną metodę przybliżania całki

Metoda aproksymacji	Dolny zakres całkowania	Górny zakres całkowania	Wynik aproksymacji	Czas ewaluacji [s]
Prostokątów	-16	16	1.380388447043143	0.0034122499999648427
Trapezów	-39	-39	1.380388447043143	0.038671576999913665
Simpsona	-9	9	1.380388447043143	0.0017419949999020901

Wnioski:

Zestawiając otrzymane wyniki zauważalne jest fakt, iż dla zadanej dokładności najbardziej wydajna okazał się metoda Simpsona, potrzebowała też ona najkrótszego przedziału do otrzymania wyniku z wymaganą precyzja. Metoda prostokątów potrzebowała przydzielenia dwukrotnie dłuższego czasu procesor niż metoda Simpsona. Natomiast najmniej wydajna okazała się metoda trapezów, gdyż jej czas wykonania dla zadanej dokładności był ponadto 22 krotnie dłuższy niż w przypadku metody Simpsona oraz 11 krotnie dłuższy od czasu wymaganego dla metody prostokątów.

Powyższy wykres funkcji podcałkowej oraz obliczenia zostały uzyskane poprzez wykonanie poniższego programu napisanego w języku Python:

```
from matplotlib import  pyplot as plt
import numpy as np
from numpy import e, cos
from typing import Final

def fun(x) -> float:
    return e**(-x**2)*cos(x)


def draw_plot(r_min: int,r_max: float, n: int) -> None:
    d = np.linspace(r_min, r_max, n)
    plt.plot(d, fun(d), color='green')
    font = {
        'family': 'cambria',
        'color': 'black',
        'weight': 'bold',
        'style': 'italic',
        'size': 10
    }
    plt.title(
        f"Wykres funkcji podcałkowej",
        fontdict=font,
        fontsize=font['size']
    )
    plt.plot(d, fun(d), color='red')
    plt.show()


def main()->None:
    R_MIN: Final = -10
    R_MAX: Final = 10
    n = 10000
    draw_plot(R_MIN, R_MAX, n)


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

```
from typing import Any
import numpy as np
from numpy import  cos, e
from numpy import ndarray, dtype
from typing import Callable
import timeit

def fun(x: float | ndarray[Any, dtype[Any]]) -> float:
```

```
    return e**(-x**2)*cos(x)

def rectangle_method(r_min: float, r_max: float, n: int) -> float:
    width =  (r_max-r_min)/n
    integral_approx = 0
    pieces = np.linspace(r_min, r_max-width, n)
    pieces += width/2
    for x in pieces:
        integral_approx += fun(x)*width
    return integral_approx


def trapezoidal_method(r_min: float, r_max: float, n: int) -> float:
    width = (r_max-r_min)/n
    pieces = np.linspace(r_min, r_max, n+1)
    integral_approx = 0
    for i in range(1, n+1):
        integral_approx += width*(fun(pieces[i]) + fun(pieces[i-1]))/2

    return integral_approx

def simpson_method(r_min:float, r_max:float , n: int) -> float:
    width = (r_max-r_min)/n
    integral_approx = 0
    pieces = np.linspace(r_min, r_max, n+1)
    for i in range(1, n// 2 + 1):
        integral_approx += (fun(pieces[2*i-2]) + 4*fun(pieces[2*i-1]) +
fun(pieces[2*i]))
    return (integral_approx*width)/3


def approximate_integral(function: Callable,exact_res: float, eps: float) -
> tuple[float, int, int]:
    integral_approx =  0
    r_min = 0
    r_max = 0

    while abs(exact_res-integral_approx) > eps:
        r_min -= 1
        r_max += 1
        n = (r_max-r_min)*10
        integral_approx = function(r_min, r_max, n)

    return integral_approx, r_min, r_max


def get_evaluation_time(function: str, n: int) -> float:
    evaluation_time = timeit.timeit(stmt=function, globals=globals(),
number=n)
    return evaluation_time/n

def main() -> None:
     eps = 1e-16
     exact_res = np.sqrt(np.pi)/ np.e**0.25
     n = 100

     print("-------------------Rectangle method-------------------")
     rectangle_method_res, r_min, r_max =
approximate_integral(rectangle_method, exact_res, eps)
     rectangle_func_to_evaluation = "approximate_integral(rectangle_method,
np.sqrt(np.pi)/ np.e**0.25,1e-16)"
```

```
    rectangle_evaluation_time =
get_evaluation_time(rectangle_func_to_evaluation,n)
    print(f'{rectangle_evaluation_time = }')
    print(f'{rectangle_method_res = }')
    print(f'{abs(exact_res-rectangle_method_res) = }')
    print(f'{r_min = }')
    print(f'{r_max = }')
    print("--------------------Trapezoidal method--------------------")
    trapezoidal_method_res, r_min, r_max =
approximate_integral(trapezoidal_method, exact_res, eps)
    trapezoidal_func_to_evaluation =
"approximate_integral(trapezoidal_method,np.sqrt(np.pi)/ np.e**0.25,1e-16)"
    trapezoidal_evaluation_time =
get_evaluation_time(trapezoidal_func_to_evaluation, n)
    print(f'{trapezoidal_evaluation_time = }')
    print(f'{trapezoidal_method_res = }')
    print(f'{abs(exact_res-trapezoidal_method_res) = }')
    print(f'{r_min = }')
    print(f'{r_max = }')
    print("--------------------Simpson's method--------------------")
    simpson_method_res,r_min, r_max = approximate_integral(simpson_method,
exact_res, eps)
    simpson_func_to_evaluation =
"approximate_integral(simpson_method,np.sqrt(np.pi)/ np.e**0.25,1e-16)"
    simpson_evaluation_time =
get_evaluation_time(simpson_func_to_evaluation, n)
    print(f'{simpson_evaluation_time = }')
    print(f'{simpson_method_res = }')
    print(f'{abs(exact_res-simpson_method_res) = }')
    print(f'{r_min = }')
    print(f'{r_max = }')

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

b)

W przypadku obliczeń za pomocą całkowania adaptacyjnego należy z góry przyjąć dokładność obliczeń, której osiągnięcie będzie końcem programu wyliczającego zadaną całkę. W tym zadaniu za jak dokładność obliczeń przyjmuję $\varepsilon = 1 \cdot 10^{-16}$.

Ponieważ rozważana funkcja $f(x) = e^{-x^2}\, cos(x) \in C^n[\mathbb{R}]$, a zatem $f \in C^4[\mathbb{R}]$, więc możemy skorzystać z metody Simpsona.

W celu rozwiązania zadania wykonujemy następujące kroki:

1. Korzystając ze wzoru Simpsona obliczamy całkę na przyjętym przedziale:

$$\int_a^b f(x)\,dx = \frac{h}{3}[f(a) + 4f(a+h) + f(b)] - \frac{h^5}{90}f^4(\mu), \mu \in (a,b)$$

   Gdzie:

$$S(a,b) = \frac{h}{3}[f(u) + 4f(a+h) + f(b)]$$

$$\alpha = \frac{h^5}{90} f^4(\mu)\alpha$$

2. Następnie obliczamy wartości przybliżenia całki ze wzoru Simpsona na dwóch połowach wejściowego przedziału.

$$\int_a^b f(x)\,dx = \frac{h}{6}\left[f(a) + 4f\left(a + \frac{h}{2}\right) + f(a+h) + f(a+h) + 4f\left(a + \frac{3h}{2}\right) + f(b)\right]$$
$$-\frac{b-a}{180}\left(\frac{b}{2}\right)^4 f^{(4)}(\mu^*), \mu^* \in (a,b)$$

Gdzie:

$$S\left(a, \frac{\alpha+b}{2}\right) = f(a) + 4f\left(a + \frac{h}{2}\right) + f(a+h)$$
$$S\left(\frac{\alpha+b}{2}, b\right) = (a+h) + 4f\left(a + \frac{3h}{2}\right) + f(b)$$
$$\alpha = \frac{b-a}{180}\left(\frac{b}{2}\right)^4 f^{(4)}(\mu^*)$$

Przyjmujemy, że $f^{(4)}(\mu) \approx f^{(4)}(\mu^*)$ oraz podstawiamy $b - a = 2h$ wówczas:

$$\alpha = \frac{h^5}{1440}\left(\frac{b}{2}\right)^4 f^{(4)}(\mu)$$

3. Porównujemy równania otrzymane w podpunkcie 1 i 2 w celu wyznaczenia błędu α.

$$S(a,b) - \alpha \approx S\left(a, \frac{\alpha+b}{2}\right) + S\left(\frac{\alpha+b}{2}, b\right) - \frac{1}{16}\alpha \Rightarrow$$
$$\alpha = \frac{16}{15}\left[S(a,b) - S\left(a, \frac{a+b}{2}\right) - S\left(\frac{o+b}{2}, b\right)\right]$$

Uzyskujemy:

$$\left|\int_a^b f(x)\,dx - S\left(a, \frac{a+b}{2}\right) - S\left(\frac{a+b}{2}, b\right)\right| \approx \frac{1}{15}\left|\left(S(a,b) - S\left(a, \frac{a+b}{2}\right) - S\left(\frac{a+b}{2}, b\right)\right)\right|$$

gdzie lewa strona to błąd przy zastosowaniu dokładniejszego sposobu całkowania . Dążymy do tego aby błąd ten był mniejszy niż przyjęte ε. Zatem otrzymujemy poniższą nierówność:

$$\left|\left(S(a,b) - S\left(a, \frac{a+b}{2}\right) - S\left(\frac{a+b}{2}, b\right)\right)\right| < 15\,\varepsilon \quad (*)$$

4. Następnie jeśli nierówność $(*)$ jest spełniona wówczas $S\left(a, \frac{a+b}{2}\right) + S\left(\frac{a+b}{2}, b\right)$ wystarczająco dobrze przybliża naszą całkę. W przeciwnym przypadku stosujemy procedurę oceny błędu do każdego z przedziałów $\left[a, \frac{a+b}{2}\right], \left[\frac{a+b}{2}, b\right]$, w każdym z nich przyjmując $\varepsilon' = \frac{\varepsilon}{2}$. Jeśli na danym podprzedziale spełniona jest nierówność $(*)$ to kończymy dla niego obliczenia w przeciwnym przypadku powtarzamy procedurę dzielenia przedziału i wyliczania przybliżenia do momentu spełnienia warunku nie równości. Finalnie jako przybliżenia naszej całki zwracamy sumę wartości wyliczonych dla wszystkich podprzedziałów.

Dla przyjętej dokładności otrzymujemy poniższe przybliżenia zadanej całki:

$$S(f) \approx 1.380388447043143$$

Metoda aproksymacji	Dolny zakres całkowania	Górny zakres całkowania	Wynik aproksymacji	Czas ewaluacji [s]
Całkowanie adaptacyjne	-8	8	1.380388447043143	3.7794039300002624

Wnioski:

Metoda całkowania adaptacyjnego do obliczenia całki z zadaną dokładnością potrzebowała najwięcej czasu (ponad 3.78 s) ze sprawdzonych dotąd metod. Najprawdopodobniej jest to związane z charakterystyką implementacji owej metody, która wykorzystuje rekurencję do obliczania składowych wartości całki na kolejnych podprzedziałach. W związku z tym dla wysokiej dokładności będzie się to wiązało z wieloma wywołaniami funkcji dla coraz mniejszych podprzedziałów , co z kolei skutkuje większym zapotrzebowaniem na czas procesora do wykonania obliczeń.

Powyższe wyniki otrzymano wykonując poniższy kod napisany w języku Python:

```
import timeit
from numpy import e, cos,sqrt, pi,  ndarray, dtype
from typing import Any, Final, Callable

def fun(x: float | ndarray[Any, dtype[Any]]) -> float:
    return e**(-x**2)*cos(x)


def quad_simpsons_mem(f, a, b):
    m = (a+b)/2
    return m, abs(b-a)/6 *(f(a) +4*f(m) + f(b))


def _quad_asr(f, a, b, eps, whole, m):
    lm, left = quad_simpsons_mem(f, a, m)
    rm,right = quad_simpsons_mem(f, m, b)

    delta = left + right - whole
    if abs(delta) <= 15*eps:
        return  left + right + delta/15

    return _quad_asr(f, a, m, eps/2, left, lm) + _quad_asr(f, m, b, eps/2,
right, rm)



def quad_asr(f, a, b, eps):

    m,whole = quad_simpsons_mem(f, a, b)
    return  _quad_asr(f, a, b, eps, whole, m)


def calculate_integral_adaptive_method(exact_res: float, eps: float) ->
tuple[float, int, int]:
    r_min = 0
    r_max = 0
    integral_approximation = 0
    while abs(exact_res - integral_approximation) > eps:
```

```
        r_min -= 1
        r_max += 1
        integral_approximation = quad_asr(fun, r_min, r_max, eps)

    return integral_approximation, r_min, r_max



def get_evaluation_time(function: str, n: int) -> float:
    evaluation_time = timeit.timeit(stmt=function, globals=globals(),
number=n)
    return evaluation_time/n



def main() -> None:
    eps = 1e-16
    exact_res = sqrt(pi)/e**0.25
    n = 10
    adaptive_integral, r_min, r_max =
calculate_integral_adaptive_method(exact_res, eps)
    adaptive_func_to_evaluation =
"calculate_integral_adaptive_method(sqrt(pi)/e**0.25,1e-16)"
    adaptive_method_time =
get_evaluation_time(adaptive_func_to_evaluation,n)
    print(f'{adaptive_method_time = }')
    print(f'{adaptive_integral = }')
    print(f'{abs(exact_res-adaptive_integral) = }')
    print(f'{r_min = }')
    print(f'{r_max = }')


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

c)

Dla funkcji z wagą $w(x) = e^{-x^2}$ na przedziale $[-\infty, \infty]$ kwadratura Gaussa-Hermite'a ma następującą postać:

$$I(f) = \int_{-\infty}^{\infty} e^{-x^2} f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Gdzie x_i to pierwiastki n-tego wielomianu Hermite'a .

Wielomiany Hermite'a możemy uzyskać korzystając z poniższej zależności rekurencyjnej:

$$H_{n+1} = 2xH_n(x) - 2nH_{n-1}(x)$$

$$H_0(x) = 1$$

$$H_1(x) = 2x$$

Natomiast w_i dane jest poniższym wzorem;

$$w_i = \frac{2^{n+1} n! \sqrt{\pi}}{[H_n'(x_i)]^2}$$

Przeprowadziwszy obliczenia za pomocą programu w Pythonie, chcąc otrzymać obliczenia z zadaną dokładnością w celu przybliżenia całki z treści zadania, należy posłużyć się wielomian Hermite'a stopnia 17 następującej postaci:

$$H_{17}(x) = 131072x^{17} - 8912896x^{15} + 233963520x^{13} - 3041525760x^{11} + 20910489600x^{9} - 75277762560x^{7} + 131736084480x^{5} - 04097203200x^{3} + 17643225600x$$

Ponieważ w naszym przypadku $f(x) = cos(x)$, za zatem wiedząc , że funkcja ta jest parzysta do obliczeń użyję tylko dodatnich pierwiastków powyższego wielomianu oraz wyliczonych dla nich wag. Finalny wynik będzie podwojeniem tak otrzymanej wartości.

i	x_i	w_i	$f(x_i)$
1	*0.0000000000000*	*0.5309179376249*	*1.0000000000000*
2	*0.5316330013427*	*0.401264694704*	*0.861980383835*
3	*1.0676487257435*	*0.1726482976701*	*0.482185438911*
4	*1.6129243142212*	*0.0409200341498*	*-0.0421155273028*
5	*2.1735028266666*	*0.0050673499576*	*-0.5668741733620*
6	*2.7577629157039*	*0.0002986432867*	*-0.9272373005708*
7	*3.3789320911415*	*0.0000071122891*	*-0.9719669589368*
8	*4.0619466758755*	*0.0000000497708*	*-0.6055384579811*
9	*4.8713451936744*	*0.0000000000458*	*0.1582876652951*

Zatem:

$$I(f) = \int_{-\infty}^{\infty} e^{-x^2} f(x)\, dx \approx 2 \sum_{i=1}^{9} w_i f(t_i) \approx 1.380388447043143$$

Metoda aproksymacji	Dolny zakres całkowania	Górny zakres całkowania	Wynik aproksymacji	Czas ewaluacji [s]
Całkowanie adaptacyjne	-8	8	1.380388447043143	0.0011888189997989683

Wnioski:

Metoda przybliżania całki za pomocą kwadratury Gaussa-Hermite'a dla przyjętej dokładności okazała się bardzo wydajna, gdyż pozwoliła na otrzymanie wyniku już po około 0.0012 s.

Powyższe obliczenia zostały wykonane za pomocą poniższego programu napisanego w języku Python:

```
from numpy import cos, e, pi, sqrt
from scipy.special import roots_hermite
import timeit

def fun(x:float)->float:
    return cos(x)


def gauss_hermite_approximation(n: int) -> float:

    roots, weights = roots_hermite(n)
    f_r = [fun(r) for r in roots]
```

```
    return sum(f_r[i]*weights[i] for i in range(len(roots)))



def calculate_integral_approximation(eps: float, exact_res: float) ->
tuple[float, int]:
    n = 0
    integral_approximation = 0
    while abs(exact_res-integral_approximation) >= eps:
        n += 1
        integral_approximation = gauss_hermite_approximation(n)

    return integral_approximation, n


def get_evaluation_time(function_name: str, n: int) -> float:
    return timeit.timeit(stmt=function_name, globals=globals(), number=n)/n


def main() -> None:
    eps = 1e-16
    exact_res = sqrt(pi) / e ** 0.25
    num = 100
    integral_approximation, n =  calculate_integral_approximation(eps,
exact_res)
    fun_name_to_evaluation = "calculate_integral_approximation(1e-16,
sqrt(pi) / e ** 0.25)"
    gauss_hermite_time = get_evaluation_time(fun_name_to_evaluation,num)
    print(f'{gauss_hermite_time = }')
    print(f'{integral_approximation = }')
    print(f'{n = }')


if __name__ == '__main__':

    main()
```

Podsumowanie:

Zestawiając wszystkie metody dla których zostały przeprowadzone obliczenia najbardziej wydajna okazała się metoda kwadratury Gaussa-Hermite'a, zaś najmniej metoda całkowania adaptacyjnego. Kolejno na podium pod względem wydajności uplasowały się kolejno metoda Simpsona i prostokątów. Natomiast metoda trapezów okazał bardziej optymalna jedynie od wypadającej najsłabiej metody całkowania adaptacyjnego.

3.Bibliografia

1. Włodzimierz Funika Materiały ze strony
2. Katarzyna Rycerz Wykład z przedmiotu Metody Obliczeniowe w Nauce i Technice
3. https://www.wolframalpha.com
4. https://en.wikipedia.org/wiki/Adaptive_Simpson%27s_method
5. https://en.wikipedia.org/wiki/Hermite_polynomials
6. https://en.wikipedia.org/wiki/Gauss%E2%80%93Hermite_quadrature
7. https://nvlpubs.nist.gov/nistpubs/jres/048/jresv48n2p111_A1b.pdf
8. Robert E. Greenwood and J. J. Miller „Zeros of the Hermite polynomials and weights for Gauss' mechanical quadrature formula"