Laboratorium 6 - Całkowanie numeryczne II

Piotr Karamon

21.04.2024r.

Treści zadań

Obliczyć przybliżoną wartość całki:

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

1. przy pomocy złożonych kwadratur (prostokątów, trapezów, Simpsona),
2. przy pomocy całkowania adaptacyjnego,
3. przy pomocy kwadratury Gaussa-Hermite'a, obliczając wartości węzłów i wag.

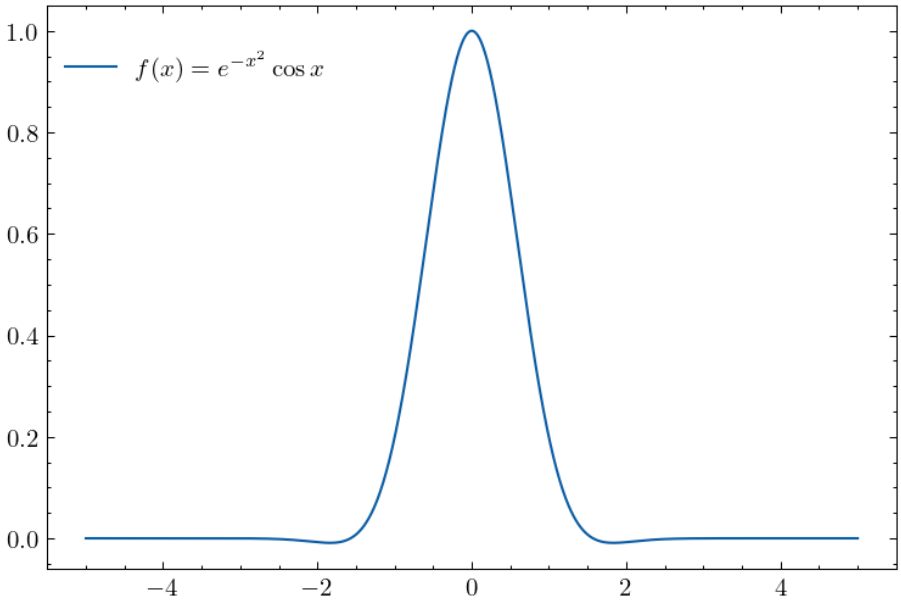
Porównać wydajność dla zadanej dokładności.

Proste kwadratury

Funkcja którą mamy scałkować ma wzór:

$$f(x) = e^{-x^2} \cos x$$

Narysujmy jej wykres



Rysunek 1: Wykres funkcji $f(x)$.

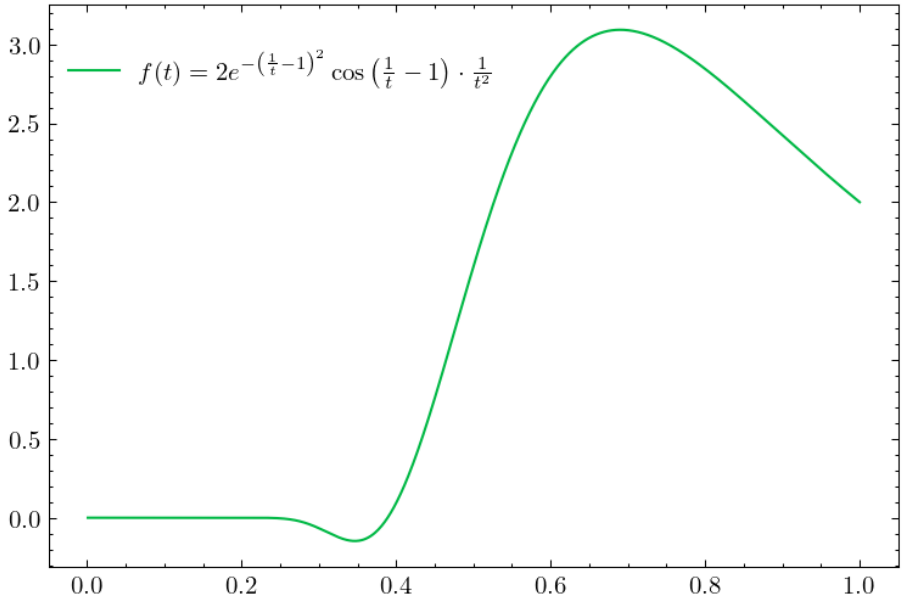
Widzimy, że wartości funkcji bardzo szybko zbiegają do zera, dzięki czynnikowi e^{-x^2}. Nasza funkcja jest parzysta zatem zamiast liczyć oryginalną całkę możemy, ją przekształcić.

$$\int_{-\infty}^{\infty} e^{-x^2}\cos(x)dx = 2\int_{0}^{\infty} e^{-x^2}\cos(x)dx =$$

Stosujemy dosyć uniwersalne podstawienie $t = \frac{1}{1+x}$

$$= \int_0^1 2e^{-\left(\frac{1}{t}-1\right)^2}\cos\left(\frac{1}{t}-1\right)\cdot\frac{1}{t^2}dt$$

Narysujemy teraz wykres funkcji podcałkowej $f(t)$



Rysunek 2: Wykres funkcji $f(t)$.

W celu porównania metod obliczmy dokładną wartość tej całki przy użyciu biblioteki Sympy.

```
from sympy import symbols, integrate, E, cos, oo

x = symbols('x')
fx = E ** (-x**2) * cos(x)
fx_int = integrate(fx, (x,-oo, oo))
print(fx_int)
print(fx_int.evalf())
```

```
sqrt(pi)*exp(-1/4)
1.38038844704314
```

Czyli dokładna wartość całki jest równa $I = \sqrt{\pi}\exp\left(-\frac{1}{4}\right)$

Metoda prostokątów

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

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

```
import numpy as np

INTEGRAL = 1.38038844704314

def f(t):
    return 2*np.e**(-(1/t-1)**2) * np.cos(1/t-1)*1/t**2

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
```

Tablica 1: Wyniki dla złożonej kwadratury prostokątów

n	S(f)	Abs Error	Relative Error [%]
1	1.5901288828	0.2097404357	15.1943053549
2	1.5013067649	0.1209183178	8.7597312253
5	1.4074352856	0.0270468386	1.9593643111
10	1.3816902507	0.0013018036	0.0943070513
20	1.3808055337	0.0004170867	0.0302151676
50	1.3804551184	0.0000666713	0.0048298959
100	1.3804051140	0.0000166670	0.0012074107
200	1.3803926137	0.0000041667	0.0003018487
500	1.3803891137	0.0000006667	0.0000482956
1000	1.3803886137	0.0000001667	0.0000120739

Metoda trapezów

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

```
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
```

By uniknąć dzielenia przez zero a ustawiamy na liczbę bardzo bliską zera (`1e-10`).

Tablica 2: Wyniki dla złożonej kwadratury trapezów

n	S(f)	Abs Error	Relative Error [%]
1	0.9999999999	0.3803884471	27.5566234967
2	1.2950644417	0.0853240054	6.1811590472
5	1.3474028839	0.0329855631	2.3895855689
10	1.3774190848	0.0029693623	0.2151106297
20	1.3795546677	0.0008337793	0.0604017891
50	1.3802551084	0.0001333387	0.0096595026
100	1.3803551134	0.0000333337	0.0024148034
200	1.3803801137	0.0000083334	0.0006036963
500	1.3803871137	0.0000013333	0.0000965912
1000	1.3803881137	0.0000003333	0.0000241478

Metoda Simpsona

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

```
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
```

Podobnie jak w przypadku trapezów zamiast $a = 0$ używamy `1e-10`

Tablica 3: Wyniki dla złożonej kwadratury Simpsona

n	S(f)	Abs Error	Relative Error [%]
1	1.3934192556	0.0130308085	0.9439957693
2	1.3934192556	0.0130308085	0.9439957693
5	1.3936081633	0.0132197162	0.9576808810
10	1.3874244850	0.0070360380	0.5097143501
20	1.3802665287	0.0001219183	0.0088321756
50	1.3803884726	0.0000000256	0.0000018527
100	1.3803884484	0.0000000013	0.0000000964
200	1.3803884471	0.0000000001	0.0000000060
500	1.3803884470	0.0000000000	0.0000000002
1000	1.3803884470	0.0000000000	0.0000000000

Wnioski: Z pośród prostych kwadratur zdecydowanie najlepsze wyniki otrzymujemy w przypadku metody Simpsona. Ta metoda nawet dla małych n daje dobre przybliżenie. Metody prostokątów oraz trapezów daję zadawalające rezultaty ale tylko gdy n jest duże, dla małych ilości przedziałów błędy są zdecydowanie za duże. Wszystkie trzy metody są bardzo łatwe w implementacji, a zatem jeżeli chcemy mieć prosty algorytm liczący dosyć dokładnie całki dla wielu funkcji to powinniśmy wybrać metodę Simpsona.

Całkowanie adaptacyjne

Będziemy całkować adaptacyjnie przy użyciu kwadratury Simpsona. Mamy zadane Δ, czyli dokładność z jaką chcemy obliczyć całkę. $S(a,b)$ oznacza wartość kwadratury Simpsona dla przedziału (a,b) Będziemy postępować w następujący sposób:

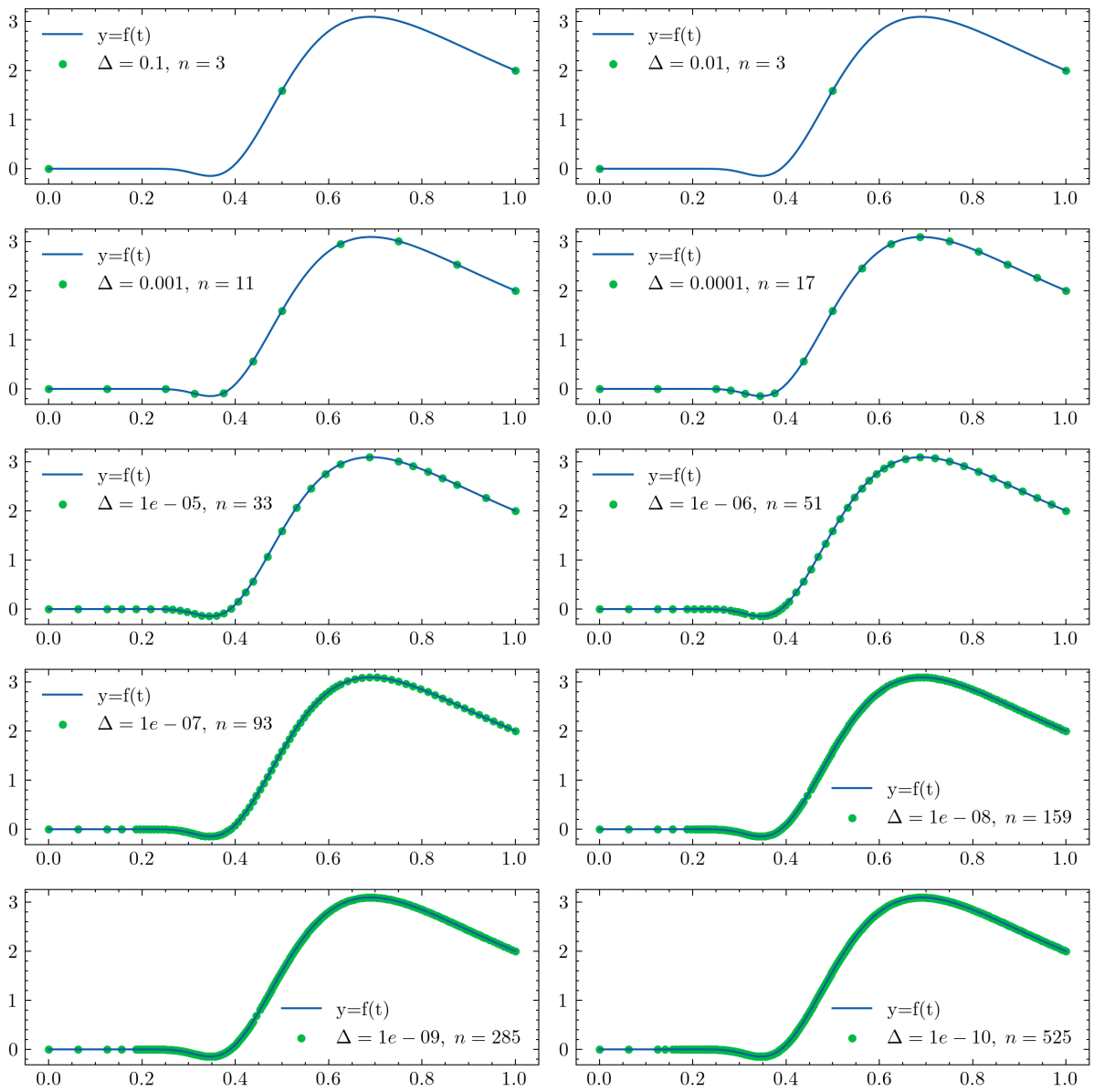
1. Jeżeli $|S(a,b) - S(a,\frac{a+b}{2}) - S(\frac{a+b}{2},b)| < 15\Delta$, to kończymy.
2. Jeżeli powyższa nierówność nie jest spełniona to stosujemy procedurę rekurencyjnie do przedziałów $[a,\frac{a+b}{2}]$, $[\frac{a+b}{2},b]$ w każdym z nich $\Delta' = \frac{\Delta}{2}$.

```python
def adaptive_integral(f, a, b, delta=1e-5, max_level=10):
    initial_estimate = simpson_quadrature(f, a, b)
    points = [a]
    result = recursive_integral(f, a, b, initial_estimate, delta, 0, max_level, points)
    points.append(b)
    return result, np.array(points)


def simpson_quadrature(f, a, b):
    h = (b-a)/2
    return  h/3 * (f(a) + 4*f(a+h) + f(b))

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

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

    points.append((a+b)/2)

    if abs(S1 + S2 - S) < 15*delta:
        return S1 + S2
    else:
        left = recursive_integral(f, a, (a + b) / 2, S1, delta / 2, level + 1, max_level, points)
        right = recursive_integral(f, (a + b) / 2, b, S2, delta / 2, level + 1, max_level, points)
        return left + right
```

Tablica 4: Wyniki dla całkowania adaptacyjnego przy użyciu kwadratury Simpsona.

Δ	Approx	n	Abs Error	Relative Error [%]
0.1000000000	1.4325593237	3	0.0521708766	3.7794344584
0.0100000000	1.4325593237	3	0.0521708766	3.7794344584
0.0010000000	1.3804846333	11	0.0000961863	0.0069680574
0.0001000000	1.3803422211	17	0.0000462260	0.0033487668
0.0000100000	1.3803879528	33	0.0000004943	0.0000358053
0.0000010000	1.3803883542	51	0.0000000929	0.0000067277
0.0000001000	1.3803884383	93	0.0000000087	0.0000006336
0.0000000100	1.3803884462	159	0.0000000008	0.0000000583
0.0000000010	1.3803884466	285	0.0000000004	0.0000000286
0.0000000001	1.3803884470	525	0.0000000000	0.0000000004

n oznacza ilość puntków w których to obliczyliśmy wartość funkcji $f(t)$



Rysunek 3: W zielonych punktach obliczana była wartość funkcji w wyniku całkowania adaptacyjnego.

Wnioski: Metoda całkowania adaptycjnego daje imponujące wyniki. Przy niewielkich ilościach przedziałów błąd zbiega szybko do zera. Na wykresach widzimy również "adaptacyjność" tej metody. Spoglądając na wykres dla $\Delta = 1e{-}5$ widzimy, że największe zagęszczenie węzłów jest przy minimum lokalnym w okolicy 0.4, ponieważ funkcja przyjmuje tam dość złożnych kształt. Natomiast w przedziale $[0, 0.2]$ funkcja jest bliska linii prostej i w tym przedziale jest niewiele węzłów, ponieważ jest to prosty kszałt który łatwo jest przybliżyć. Ogromną zaletą tej metody jest możliwość wybrania dokładności z jaką chcemy otrzymać wartość całki.

Kwadrautra Gaussa-Hermite'a

Wielomiany Hermite'a to wieloniamy spełniające równanie

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

Spełniają one rekurecyjne równanie:

$$H_0(x) = 1$$
$$H_{n+1}(x) = 2xH_n(x) - H_n'(x)$$

Używając biblioteki Sympy możemy łatwo wyznaczyć wzory tych wielomianów

```
from sympy import symbols,lambdify, diff, expand, latex
x = symbols('x')

def hermite(n):
    if n == 0:
        return 1
    lower = hermite(n-1)
    return x*2*lower - diff(lower, x)
```

$$\begin{aligned}
H_0(x) &= 1 \\
H_1(x) &= 2x \\
H_2(x) &= 4x^2 - 2 \\
H_3(x) &= 8x^3 - 12x \\
H_4(x) &= 16x^4 - 48x^2 + 12 \\
H_5(x) &= 32x^5 - 160x^3 + 120x \\
H_6(x) &= 64x^6 - 480x^4 + 720x^2 - 120 \\
H_7(x) &= 128x^7 - 1344x^5 + 3360x^3 - 1680x \\
H_8(x) &= 256x^8 - 3584x^6 + 13440x^4 - 13440x^2 + 1680 \\
H_9(x) &= 512x^9 - 9216x^7 + 48384x^5 - 80640x^3 + 30240x \\
H_{10}(x) &= 1024x^{10} - 23040x^8 + 161280x^6 - 403200x^4 + 302400x^2 - 30240 \\
H_{11}(x) &= 2048x^{11} - 56320x^9 + 506880x^7 - 1774080x^5 + 2217600x^3 - 665280x
\end{aligned}$$

Wielomiany te są ortogonalne z funkcją wagową $w(x) = e^{-x^2}$, czyli

$$\int_{-\infty}^{\infty} H_m(x)H_n(x)e^{-x^2}dx = 0, \quad \text{dla} \quad m \neq e$$

Będziemy postępować w następujący sposób:

1. Dla zadanego n znajdziemy miejsca zerowe $H_n(x)$ numerycznie funkcją biblioteczną.
2. Rozwiążemy układ równań w celu uzyskania wag

$$\begin{bmatrix} H_0(x_1) & \cdots & H_0(x_n) \\ \vdots & \ddots & \vdots \\ H_{n-2}(x_1) & \cdots & H_{n-2}(x_n) \\ H_{n-1}(x_1) & \cdots & H_{n-1}(x_n) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \int_{-\infty}^{\infty} w(x)H_0(x)\,dx = \sqrt{\pi} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

3. Wartość całki policzymy ze wzoru:

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

Tworzymy funkcję która obliczy miejsca zerowe oraz potrzebne wagi.

```python
import numpy as np
from numpy.polynomial import  hermite
from numpy.polynomial.hermite import hermroots, Hermite

def hermite_roots_weights(n):
    roots = hermroots([0]*n +[1])
    Hs = [Hermite([0] * i + [1]) for i in range(0, n)]

    A = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j] = Hs[i](roots[j])
    b = np.zeros(n)
    b[0] = np.sqrt(np.pi)

    weights = np.linalg.solve(A, b)
    return np.array(roots), np.array(weights)
```

Tablica 5: Wagi i węzły dla $n = 5$

i	x_i	w_i
1	-2.0201828705	0.0199532421
2	-0.9585724646	0.3936193232
3	0.0000000000	0.9453087205
4	0.9585724646	0.3936193232
5	2.0201828705	0.0199532421

I funkcję która oblicza całkę

```python
def hermite_approx(f, n):
    roots, weights = hermite_roots_weights(n)
    return np.sum(f(roots) * weights)
```

Tablica 6: Wyniki dla całkowania kwadraturą Gaussa-Hermite'a.

n	Approx	Abs Error	Relative Error [%]
1	1.772453850906	0.392065403862	28.402541668774
2	1.347498463717	0.032889983326	2.382661445536
3	1.382033071388	0.001644624345	0.119142140637
4	1.380329757161	0.000058689882	0.004251693211
5	1.380390075936	0.000001628893	0.000118002474
6	1.380388410051	0.000000036992	0.000002679855
7	1.380388447754	0.000000000711	0.000000051503
8	1.380388447031	0.000000000012	0.000000000858
9	1.380388447043	0.000000000000	0.000000000013
10	1.380388447043	0.000000000000	0.000000000000

Wnioski: Widzimy ogromną przewagę kwadratury Gaussa-Hermite'a nad innymi metodami. Dokładność podobną jak dla $n = 10$ w tej metodzie możemy zaobserwować w poprzednich sposobach dla $n = 1000$. Co sprawia, że po wyliczeniu miejsc zerowych oraz wag, kwadratura Gaussa-Hermite'a jest znaczenie bardziej oszczędna obliczeniowo. Błąd względny na poziomie poniżej 1% widzimy już przy $n = 3$. Zaletą tej

metody jest również to, że świetnie działa ona na przedziale $[-\infty, \infty]$ bez żadnych przekształceń. Wadą natomiast jest mała uniwersalność.

Podsumowanie

Analiza porównawcza prostych kwadratur, całkowania adaptacyjnego i kwadratury Gaussa-Hermite'a potwierdza ich różnorodną efektywność dla zadanego problemu całkowania. Metoda Simpsona, charakteryzująca się wyjątkową dokładnością nawet przy mniejszej liczbie podziałów wykazuje się najlepszą efektywnością wśród prostych kwadratur. Całkowanie adaptacyjne, znacząco zwiększa precyzję wyniku, szczególnie w rejonach, gdzie funkcja wykazuje skomplikowane zachowanie, przy jednoczesnym ograniczeniu liczby potrzebnych obliczeń. Natomiast kwadratura Gaussa-Hermite'a, choć wymaga wstępnego wyznaczenia miejsc zerowych i wag, okazuje się być najbardziej wydajna obliczeniowo, gwarantując wysoką dokładność przy znacznie niższym koszcie obliczeniowym.

Bibliografia

- Marian Bubak, Katarzyna Rycerz: *Metody Obliczeniowe w Nauce i Technice. Kwadratury*
- Gauss-Hermite Quadrature
- Hermite polynomials