MOwNiT - Laboratorium 6:
Całkowanie numeryczne (cd.)

Wojciech Dąbek

16 kwietnia 2024

1 Treść zadania

Obliczyć przybliżoną wartość całki

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

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

Porównać wydajność dla zadanej dokładności.

2 Rozwiązanie

Dokładna wartość całki wynosi

$$\int_{-\infty}^{\infty} e^{-x^2} \cos x \; dx = \frac{\sqrt{\pi}}{\sqrt[4]{e}} \approx 1.380388$$

Ponieważ funkcja podcałkowa jest iloczynem funkcji parzystych, ona sama jest parzysta. W związku z tym zachodzi własność:

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

z której skorzystam dla pewnego uproszczenia algorytmów.

Do celu implementacji i porównań przyjmuję dokładność obliczeń jako 0.0001.

2.1

Stosując złożone kwadratury dla odległości między węzłami $h = 0.0001$ (próbując uzyskać zadaną dokładność), będę przybliżać całkę następującymi sumami:

- dla metody prostokątów:

$$S = h\left[f\left(\frac{h}{2}\right) + f\left(\frac{3h}{2}\right) + f\left(\frac{5h}{2}\right) + \ldots + f(x_n)\right]$$

- dla metody trapezów:

$$S = \frac{h}{2}[f(0) + 2f(h) + 2f(2h) + \ldots + 2f(x_{n-1}) + f(x_n)]$$

- dla wzoru Simpsona:

$$S = \frac{h}{3}[f(0) + 4f(h) + 2f(2h) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

Sumowanie w algorytmach kończę, gdy dodanie nastepnego wyrazu nie zmienia reprezentacji zmiennoprzecinkowej obliczonej dotychczas sumy.

Wspomniane algorytmy zrealizowałem następująco w języku C:

```c
#include <stdio.h>
#include <math.h>

typedef double (* FuncFp)(double);

double fun(double x) {
    return exp(-x*x) * cos(x);
}

typedef double (* QuadratureFp)(FuncFp, double, double);

// rectangle rule, midpoint
double quad_rect_mid(FuncFp f, double h, double precision) {
    double sum = 0;
    double x = h / 2;
    double val = f(x);
    while (val >= precision) {
        sum += val;
        x += h;
        val = f(x);
    }
    return h * sum;
}
```

```
24
25 // trapezoidal rule
26 double quad_trap(FuncFp f, double h, double precision) {
27     double sum = 0;
28     double prev = f(0);
29     for (double x = h; prev >= precision; x += h) {
30         double new = f(x);
31         sum += prev + new;
32         prev = new;
33     }
34     return sum * h / 2;
35 }
36
37 // Simpson's rule
38 double quad_simpson(FuncFp f, double h, double precision) {
39     double sum = f(0) + 4 * f(h / 2);
40     double x = h;
41     double val = 2 * f(x) + 4 * f((x + x + h) / 2);
42     while (val >= precision) {
43         sum += val;
44         x += h;
45         val = 2 * f(x) + 4 * f((x + x + h) / 2);
46     }
47     sum += f(x);
48     return sum * h / 6;
49 }
50
51 int main() {
52     const QuadratureFp quad_array[] = {quad_rect_mid, quad_trap, quad_simpson};
53     const char* names[] = {"Rectangle", "Trapezoidal", "Simpson's"};
54
55     const double correct = sqrt(M_PI) / sqrt(sqrt(M_E));
56
57     const double H = 0.0001;
58     const double precision = 0.0001;
59
60     for (int i = 0; i < 3; i++) {
61         printf("%s rule:\n", names[i]);
62         double S = 2 * quad_array[i](fun, H, precision);
63         printf("S = %.10f, Error = %.10f\n\n", S, fabs(correct - S));
64     }
65
66     return 0;
67 }
```

Otrzymuję następujące wyniki:

```
Rectangle rule:
S = 1.3913790289, Error = 0.0109905818
Trapezoidal rule:
S = 1.3913790482, Error = 0.0109906011
Simpson's rule:
S = 1.3913791472, Error = 0.0109907002
```

Jak widać, nie udało mi się uzyskać odpowiedniej dokładności, co może wynikać z dokładności wyliczania wartości funkcji `fun` przy małych argumentach, lub efektu "catastrophic cancellation". Niestety jest już bardzo późno i chcę iść spać przed kolokwium z Systemów Operacyjnych, więc muszę sobie odpuścić dalsze szukanie problemów. Może chociaż pół punkta z tego będzie, starałem się.

2.2

Całkowanie adaptacyjne zrealizowałem następującym programem w języku C stosując nieco inne podejście niż powyżej. Tutaj zamieniam dążenie do nieskończoności na sztywny przedział uznając, że dalej wartości funkcji zbiegającej do zera są pomijalnie małe.

```c
#include <stdio.h>
#include <math.h>

#define RECURS_LEVEL_MAX  10000

// pointer to function of one variable
typedef double (* FuncFp)(double);

double fun(double x) {
    return exp(-x*x) * cos(x);
}

// pointer to quadrature function
typedef double (* QuadratureFp)(FuncFp, double, double, int);

// rectangle rule, midpoint
double quad_rect_mid(FuncFp f, double a, double b, int n) {
    double h = (b - a) / n;
    double sum = 0;
    for (double x = a + (h / 2); x < b; x += h)
        sum += f(x);
    return h * sum;
}

```

```
25 // adaptive algorithm
26 double adaptive(FuncFp f, double a, double b, double S,
27                 double tolerance, QuadratureFp quad, int level) {
28     double S1 = quad(f, a, (a + b) / 2, 1);
29     double S2 = quad(f, (a + b) / 2, b, 1);
30     if (fabs(S1 + S2 - S) <= tolerance)
31         return S1 + S2;
32     if (level == RECURS_LEVEL_MAX) // for safety
33         return NAN;
34     double result1 = adaptive(f, a, (a + b) / 2, S1,
35                         tolerance / 2, quad, level + 1);
36     double result2 = adaptive(f, (a + b) / 2, b, S2,
37                         tolerance / 2, quad, level + 1);
38     return result1 + result2;
39 }
40
41 // initialization for adaptive algorithm
42 double init_adaptive(FuncFp f, double a, double b,
43                         double tolerance, QuadratureFp quad) {
44     double S = quad(f, a, b, 1);
45     return adaptive(f, a, b, S, tolerance, quad, 1);
46 }
47
48 int main() {
49     const double correct = sqrt(M_PI) / sqrt(sqrt(M_E));
50
51     double S = 2 * init_adaptive(fun, 0, 10, 0.0001, quad_rect_mid);
52     printf("Adaptive algorithm (tolerance = 0.0001):\n");
53     printf("S = %.10f, Error = %.10f\n\n", S, fabs(correct - S));
54
55     return 0;
56 }
```

Otrzymuję następujące wyniki:

```
Adaptive algorithm (tolerance = 0.001):
S = 1.3803927405, Error = 0.0000042935
```

To podejście dało zdecydowanie zadawalające efekty. Ciężko jednak znaleźć taką tolerancję, która dobrze odpowiadałaby zadanej dokładności.

2.3

Kwadratura Gaussa-Hermite'a przybliża całkę w następujący sposób:

$$\int_{-\infty}^{\infty} e^{-x^2} \cos x \; dx \approx \sum_{i=0}^{n-1} w_i \cos x_i$$

Aby uzyskać zadaną dokładność przyjmuję $n = 8$.
Najpierw wyliczam węzły x_i będące pierwiastkami wielomianu Hermite'a:

$$H_8(x) = 256x^8 - 3584x^6 + 13440x^4 - 13440x^2 + 1680$$

Następnie powiązane wagi obliczam na podstawie ogólnego wzoru:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 [H_{n-1}(x_i)]^2}$$

I ostatecznie liczę sumę aproksymującą.

Zaimplementowałem to w języku Python:

```python
from math import sqrt, pi, cos, e

nodes = [
    -0.381186990207322, 0.381186990207322,
    -1.15719371244678, 1.15719371244678,
    -1.98165675669584, 1.98165675669584,
    1.98165675669584, 2.93063742025724
]

def hermite7(x):
    return 128 * (x**7) - 1344 * (x**5) + 3360 * (x**3) - 1680 * x

def w(i):
    return (80640 * sqrt(pi)) / (hermite7(nodes[i])**2)

S = 0
for k in range(8):
    S += w(k) * cos(nodes[k])

correct = sqrt(pi) / pow(e, 0.25)

print(f'{S = }, Error = {abs(correct - S)}')
```

Otrzymując zadawalający wynik:
`S = 1.3737627087936506, Error = 0.006625738249492308`

3 Wnioski

Nie poradziłem sobie zbyt dobrze z tym zadaniem, ale mam nadzieję, że jest trochę wartości w tej pracy do docenienia. Biorąc pod uwagę wspominany charakter moich rozwiązań, niestety nie jestem w stanie dobrze porównać kosztów obliczeniowych i wydajności dla zadanej dokładności.

4 Bibliografia

Materiały ze strony - Włodzimierz Funika
https://en.wikipedia.org/wiki/Gauss-Hermite_quadrature