MOwNiT - Laboratorium 5: Całkowanie numeryczne

Wojciech Dąbek

9 kwietnia 2024

1 Treści zadań laboratoryjnych

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 $n = 8$.

2 Treści zadań domowych

1. Obliczyć całkę $I = \int_0^1 \frac{1}{1+x^2} dx$ 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.

3 Rozwiązania zadań laboratoryjnych

3.1

Rozważana całka jest równa

$$I = \int_0^1 \frac{1}{1+x} dx = \log 2 \approx 0.69314718$$

Dla uproszczenia zapisu przyjmuję $h = \frac{b-a}{n} = \frac{1}{n}$, x_i - granice przedziałów (np. $0, \frac{1}{3}, \frac{2}{3}, 1$ dla $n = 3$). Następujące wyniki i błędy otrzymujemy całkując dla $N = 1,\ 3,\ 5$ metodą

- prostokątów:

$$S(f) = h \sum_{i=1}^{n-1} f(x_i + \frac{h}{2})$$

- trapezów:

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

- wzoru Simpsona:

$$S(f) = \frac{h}{3} \sum_{i=1}^{n/2} [f(x_{2i-2}) + f(x_{2i-1}) + f(x_{2i})]$$

Co zrealizowałem przez poniższy program w języku C:

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

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

double fun(double x) {
    return 1. / (x + 1.);
}

// 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;
}

// trapezoidal rule
double quad_trap(FuncFp f, double a, double b, int n) {
    double h = (b - a) / n;
    double sum = 0;
    double prev = f(a);
    for (double x = a + h; x < b + h; x += h) {
        double new = f(x);
        sum += prev + new;
        prev = new;
    }
    return sum * h / 2;
}

// Simpson's rule
double quad_simpson(FuncFp f, double a, double b, int n) {
    double h = (b - a) / n;
    double sum = f(a) + 4 * f((a + a + h) / 2);
    for (double x = a + h; x < b; x += h)
        sum += 2 * f(x) + 4 * f((x + x + h) / 2);
    sum += f(b);
    return sum * h / 6;
}
```

```
45
46 int main() {
47     const int N_array[] = {1, 3, 5};
48     const QuadratureFp quad_array[] =
49         {quad_rect_mid, quad_trap, quad_simpson};
50     const char* names[] =
51         {"Rectangle", "Trapezoidal", "Simpson's"};
52
53     const double correct = log(2);
54
55     for (int i = 0; i < 3; i++) {
56         printf("%s rule:\n", names[i]);
57         for (int n = 0; n < 3; n++) {
58             double S = quad_array[i](fun, 0, 1, N_array[n]);
59             printf("n = %d: S = %.10f, Error = %.10f\n",
60                     N_array[n], S, fabs(correct - S));
61         }
62         printf("\n");
63     }
64
65     return 0;
66 }
```

Wypisuje on następujące rezultaty:

```
Rectangle rule:
n = 1: S = 0.6666666667, Error = 0.0264805139
n = 3: S = 0.6897546898, Error = 0.0033924908
n = 5: S = 0.6919078857, Error = 0.0012392948

Trapezoidal rule:
n = 1: S = 0.7500000000, Error = 0.0568528194
n = 3: S = 0.7000000000, Error = 0.0068528194
n = 5: S = 0.6956349206, Error = 0.0024877401

Simpson's rule:
n = 1: S = 0.6944444444, Error = 0.0012972639
n = 3: S = 0.6931697932, Error = 0.0000226126
n = 5: S = 0.6931502307, Error = 0.0000030501
```

Wnioski: W każdym przypadku zwiększenie liczby podprzedziałów daje wyraźnie dokładniejsze wyniki. Metody prostokątów i trapezów skutkują podobną dokładnością. Niezwykle pozytywnie wyróżnia się tu wzór Simpsona.

3.2

Dla porównania policzę najpierw dokładną wartość całki:

$$\int_{-1}^{1} \frac{1}{1+x^2} dx = \frac{\pi}{2} \approx 1.5707963$$

Mam do czynienia z przedziałem skończonym $[-1, 1]$, więc posłużę się następującym ciągiem ortogonalnych wielomianów Legendre'a:

$$P_n(x) = \frac{1}{2^n \cdot n!} \frac{d^n}{dx^n}(x^2 - 1)^n$$

Do zastosowania tego typu kwadratury Gaussa-Legendre'a przyjmuję $p(x) = 1$. Kwadratur wysokiego rzędu używa się rzadko, więc stablicowane są tylko wartości węzłów i wag dla niskich rzędów. Do uzyskania takiej tablicy wartości dla $n = 8$ posłużyłem się kalkulatorem portalu *efunda* otrzymując:

k	węzły x_k	wagi w_k
1	-0.96029	0.101229
2	-0.796667	0.222381
3	-0.525532	0.313707
4	-0.183435	0.362684
5	0.183435	0.362684
6	0.525532	0.313707
7	0.796667	0.222381
8	0.96029	0.101229

Pozostaje wstawić te wartości do wzoru:

$$\int_{-1}^{1} \frac{1}{1+x^2} dx \approx \; S = \sum_{k=1}^{8} w_k \frac{1}{1+x_k^2} = 1.5707957502350354$$

$$\Delta S = \left| \frac{\pi}{2} - S \right| \approx 5.765598611873202 \cdot 10^{-7}$$

Co obliczyłem poniższym programem napisanym w języku Erlang.

```
-module(gauss).
-export([result/0]).

fun1(X) -> 1 / (1 + X*X).

nodes_and_weights(8) -> [
  {-0.96029, 0.101229}, {-0.796667, 0.222381},
  {-0.525532, 0.313707}, {-0.183435, 0.362684},
  {0.183435, 0.362684}, {0.525532, 0.313707},
  {0.796667, 0.222381}, {0.96029, 0.101229}
].

result() ->
  Summer = fun ({Node, Weight}, Acc) ->
                Acc + Weight * fun1(Node) end,
  S = lists:foldl(Summer, 0, nodes_and_weights(8)),
  io:format("Quadrature result: ~w   Error: ~w~n",
            [S, abs(math:pi()/2 - S)]).
```

Wnioski: Ta metoda pozwala uzyskać bardzo dokładne wyniki niskim kosztem obliczeniowym w bardzo prosty sposób. Dla wyższych rzędów trudnością może być znalezienie węzłów i wag, więc w praktyce często warto zastosować złożenie obliczeń dla niższych rzędów.

4 Rozwiązania zadań domowych

4.1

Do obliczeń zastosowałem poniższy program w języku C. W całkowaniu adaptacyjnym zastosowałem wzór prostokątów.

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

#define RECURS_LEVEL_MAX  100

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

double fun(double x) {
    return 1. / (x*x + 1);
}

// 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;
}

// adaptive algorithm
double adaptive(FuncFp f, double a, double b, double S,
        double tolerance, QuadratureFp quad, int level) {
    double S1 = quad(f, a, (a + b) / 2, 1);
    double S2 = quad(f, (a + b) / 2, b, 1);
    if (fabs(S1 + S2 - S) <= tolerance)
        return S1 + S2;
    if (level == RECURS_LEVEL_MAX) // for safety
        return NAN;
    double result1 = adaptive(f, a, (a + b) / 2, S1,
                        tolerance / 2, quad, level + 1);
    double result2 = adaptive(f, (a + b) / 2, b, S2,
                        tolerance / 2, quad, level + 1);
    return result1 + result2;
}

// initialization for adaptive algorithm
double init_adaptive(FuncFp f, double a, double b,
                double tolerance, QuadratureFp quad) {
    double S = quad(f, a, b, 1);
    return adaptive(f, a, b, S, tolerance, quad, 1);
}
```

```c
int main() {
    const double correct = log(2);

    double S = quad_rect_mid(fun, 0, 1, 10);
    printf("Rectangle rule (h = 0.1):\n");
    printf("S = %.10f, Error = %.10f\n\n", S, fabs(correct - S));

    S = init_adaptive(fun, 0, 1, 0.1, quad_rect_mid);
    printf("Adaptive algorithm (tolerance = 0.1):\n");
    printf("S = %.10f, Error = %.10f\n\n", S, fabs(correct - S));

    S = init_adaptive(fun, 0, 1, 0.005, quad_rect_mid);
    printf("Adaptive algorithm (tolerance = 0.005):\n");
    printf("S = %.10f, Error = %.10f\n", S, fabs(correct - S));

    return 0;
}
```

Program wypisuje poniższe rezultaty:

```
Rectangle rule (h = 0.1):
S = 0.7856064963, Error = 0.0924593157

Adaptive algorithm (tolerance = 0.1):
S = 0.7905882353, Error = 0.0974410547

Adaptive algorithm (tolerance = 0.005):
S = 0.7854335733, Error = 0.0922863927
```

Wnioski: Dla mało skomplikowanych funkcji jak ta, mało skomplikowane metody jak wzór prostokątów przy odpowiedniej ilości podziałów mogą być wystarczające do uzyskania wyników podobnie zadawalających jak metody adaptacyjnej (która dokładniejszy wynik daje dopiero przy wyraźnie niskiej tolerancji na błędy).

4.2

Policzę najpierw dokładną wartość zadanej całki:

$$\int_0^1 \frac{1}{x+3}dx = \log\frac{4}{3} \approx 0.28768207$$

Aby móc zastosować wielomiany Legendre'a muszę najpierw przekształcić całkę na przedział $[-1, 1]$:

$$\int_0^1 \frac{1}{x+3}dx = \frac{1-0}{2}\int_{-1}^1 \frac{1}{\left(\frac{1-0}{2}x + \frac{1+0}{2}\right)+3}dx = \int_{-1}^1 \frac{1}{x+7}dx$$

Tabela przybliżonych wartości węzłów i wag dla $n = 4$ kwadratury Gaussa-Legendre'a wygląda następująco:

k	węzły x_k	wagi w_k
1	-0.861136	0.347855
2	-0.339981	0.652145
3	0.339981	0.652145
4	0.861136	0.347855

Wstawiam do wzoru kwadratury i obliczam:

$$\int_{-1}^{1} \frac{1}{x+7} dx \approx S = \sum_{k=1}^{4} w_k \frac{1}{x_k + 7} = 0.2876820714904883$$

$$\Delta S = \left| \log \frac{4}{3} - S \right| \approx 9.612925455648735 \cdot 10^{-10}$$

Do czego posłużyłem się podobnie jak wyżej programem w Erlangu.

```
-module(gauss).
-export([result/0]).

fun2(X) -> 1 / (7 + X).

nodes_and_weights(4) -> [
  {-0.861136, 0.347855},
  {-0.339981, 0.652145},
  {0.339981, 0.652145},
  {0.861136, 0.347855}
].

result() ->
  Summer = fun ({Node, Weight}, Acc) ->
                Acc + Weight * fun2(Node) end,
  S = lists:foldl(Summer, 0, nodes_and_weights(4)),
  io:format("Quadrature result: ~w   Error: ~w~n",
            [S, abs(math:log(4/3) - S)]).
```

5 Bibliografia

Materiały ze strony - Włodzimierz Funika
https://www.efunda.com/math/num_integration/findgausslegendre.cfm
https://en.wikipedia.org/wiki/Gaussian_quadrature