Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

ОТЧЕТ

по лабораторной работе № 2

по курсу «Численные методы»

на тему: «Приближённое вычисление определённого интеграла»

Студент ИУ9-61Б (Группа) _____ (Подпись, дата) Афанасьев И. (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Домрачева А. Б. (И. О. Фамилия)

2024 г.

1 Постановка задачи

Найти $\int_0^1 e^x dx$ методом прямоугольников, методом трапеций и по формуле Симпсона с погрешностью $\varepsilon = 0.001$. Сравнить требуемое число разбиений для всех трёх методов.

2 Теоретический раздел

2.1 Метод прямоугольников

Метод заключается в вычислении площади под графиком подынтегральной функции с помощью суммирования площадей прямоугольников, ширина которых определяется шагом разбиения, а высота — значением подынтегральной функции в узле интегрирования.

Пусть требуется определить значение интеграла функции $f(x)$ на отрезке $[a, b]$. Тогда отрезок разбивается на n равных отрезков длиной $h = \frac{b-a}{n}$. Получаем разбиение данного отрезка точками

$$x_{i-0.5} = a + (i - 0.5)h, \quad i = 1, ..., n.$$

Тогда приближённое значение интеграла на всём отрезке будет равно:

$$I^* = h \sum_{i=1}^{n} f(x_{i-0.5}) = h \sum_{i=1}^{n} f(a + (i - 0.5)h)$$

2.2 Метод трапеций

Метод заключается в вычислении площади под графиком подынтегральной функции с помощью суммирования площадей трапеций, высота которых определяется шагом разбиения, а высота — значением подынтегральной функции в узле интегрирования.

Пусть требуется определить значения интеграла функции $f(x)$ на отрезке $[a, b]$. Тогда отрезок разбивается на n равных отрезков длиной $h = \frac{b-a}{n}$. Получаем разбиение данного отрезка точками

$$x_i = a + ih, \quad i = 1, ..., n.$$

Тогда приближённое значение интеграла на всём отрезке будет равно

$$I^* = h(\frac{f(a) + f(x_1)}{2} + \frac{f(x_1) + f(x_2)}{2} + ... + \frac{f(x_{n-1}) + f(b)}{2}) =$$

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

2.3 Метод Симпсона

Метод заключается в приближении функции на отрезке $[a,b]$ интерполяционным многочленом второй степени функции $P_2(x)$:

$$P_2(x)=f_{i-0.5}+\frac{f_i-f_{i-1}}{h}(x_i-x_{i-0.5})+\frac{f_i-2f_{i-0.5}+f_{i-1}}{\frac{h^2}{2}}(x_i-x_{i-0.5})^2$$

Тогда приближённое значение интеграла на всём отрезке равняется

$$I^*=\frac{h}{6}(f(a)+f(b)+4\sum_{i=1}^{n}f(x_{i-0.5})+2\sum_{i=1}^{n-1}f(x_i))$$

2.4 Уточнение значения интеграла по Ричардсону

Точное значение $I\approx I_h^*+\mathcal{O}(h^k)$, где k — порядок точности метода, I_h^* — приближённое значение интеграла, вычисленного с помощью метода с шагом h. Для метода прямоугольников и метода трапеций $k=2$, для метода Симпсона $k=4$. Считаем, что $\mathcal{O}(h^k)\approx ch^k$, где c — некоторая константа, h — шаг, и $I=I_h^*+ch^k$ для шага h, $I=I_{\frac{h}{2}}^*+c(\frac{h}{2})^k$ для шага $\frac{h}{2}$. Получаем

$$I=I_{\frac{h}{2}}^*+R,$$

где R — уточнение по Ричардсону:

$$R=\frac{I_{\frac{h}{2}}^*-I_h^*}{2^k-1}.$$

Для приближённого вычисления интеграла с заданной точностью ε используется правило Рунге:

$$|R|<\varepsilon$$

.

3 Практический раздел

В листинге 3.1 представлен исходный код программы на языке C++.

Листинг 3.1 – Исходный код программы на языке C++

```
#include <cassert>
#include <cmath>
#include <functional>
#include <iomanip>
#include <iostream>
#include <limits>
#include <memory>
#include <numbers>
#include <vector>

namespace {

constexpr auto kEps = 1e-3;
constexpr std::size_t kW = 15;

struct Integral {
  double a, b;
  std::function<double(const double x)> f;
  double val;
};

struct Approximation {
  std::size_t n;
  double val;
  double r;
};

class IntegrationMethod {
 public:
  virtual ~IntegrationMethod() = default;

  virtual double Accuracy() const = 0;

  virtual double Calculate(const Integral& i, const std::size_t
     n) const = 0;

  virtual std::string_view Name() const = 0;
```

```
};

class RectangleMethod final : public IntegrationMethod {
 public:
  double Accuracy() const noexcept override { return 2; }

  double Calculate(const Integral& i,
                   const std::size_t n) const noexcept override {
    const auto h = (i.b - i.a) / n;

    auto f_sum = 0.0;
    auto x = i.a + 0.5 * h;
    for (std::size_t j = 0; j < n; ++j) {
      f_sum += i.f(x);
      x += h;
    }

    return f_sum * h;
  }

  std::string_view Name() const noexcept override { return
     "Rectangle"; }
};

class TrapezoidMethod final : public IntegrationMethod {
 public:
  double Accuracy() const noexcept override { return 2; }

  double Calculate(const Integral& i,
                   const std::size_t n) const noexcept override {
    const auto h = (i.b - i.a) / n;

    auto f_sum = 0.0;
    auto x = i.a + h;
    for (std::size_t j = 1; j < n; ++j) {
      f_sum += i.f(x);
      x += h;
    }

    return h * (f_sum + 0.5 * (i.f(i.a) + i.f(i.b)));
  }
```

```
77
78    std::string_view Name() const noexcept override { return
          "Trapezoid"; }
79  };
80
81  class SimpsonMethod final : public IntegrationMethod {
82   public:
83    double Accuracy() const noexcept override { return 4; }
84
85    double Calculate(const Integral& i,
86                     const std::size_t n) const noexcept override {
87      const auto h = (i.b - i.a) / n;
88
89      auto f_sum1 = 0.0;
90      auto x = i.a + 0.5 * h;
91      for (std::size_t j = 0; j < n; ++j) {
92        f_sum1 += i.f(x);
93        x += h;
94      }
95
96      auto f_sum2 = 0.0;
97      x = i.a + h;
98      for (std::size_t j = 1; j < n; ++j) {
99        f_sum2 += i.f(x);
100       x += h;
101     }
102
103     return h / 6 * (i.f(i.a) + i.f(i.b) + 4 * f_sum1 + 2 *
            f_sum2);
104   }
105
106   std::string_view Name() const noexcept override { return
          "Simpson"; }
107 };
108
109 void PrintTable(const
        std::vector<std::unique_ptr<IntegrationMethod>>& methods,
110                 const std::vector<Approximation>& approxs) {
111   // clang-format off
112   std::cout << std::setw(kW) << "Method"
113             << std::setw(kW) << "n"
```

```
114               << std::setw(kW) << "I"
115               << std::setw(kW) << "R"
116               << std::setw(kW) << "I + R"
117               << '\n';
118
119   auto end = methods.size();
120   assert(end == approxs.size());
121
122   for (std::size_t i = 0; i < end; ++i) {
123     const auto& method = *methods[i];
124     const auto& approx = approxs[i];
125
126     std::cout << std::setw(kW) << method.Name()
127               << std::setw(kW) << approx.n
128               << std::setw(kW) << approx.val
129               << std::setw(kW) << approx.r
130               << std::setw(kW) << approx.val + approx.r
131               << '\n';
132   }
133   // clang-format on
134 }
135
136 double CalculateRichardson(const double val, const double
      val_freq,
137                            const std::size_t k) {
138   return (val_freq - val) / ((2 << k) - 1);
139 }
140
141 Approximation CalculateApproximation(const Integral& i,
142                                      const IntegrationMethod&
                                           method,
143                                      const double eps) noexcept {
144   std::size_t n = 1;
145   double val_freq = method.Calculate(i, n);
146   double val, r;
147
148   do {
149     n <<= 1;
150     val = val_freq;
151     val_freq = method.Calculate(i, n);
152     r = CalculateRichardson(val, val_freq, method.Accuracy());
```

```
153   } while (std::abs(r) >= eps);
154
155   return {n, val_freq, r};
156 }
157
158 }  // namespace
159
160 int main() {
161   const Integral i{0, 1, [](const double x) { return
        std::exp(x); },
162                    std::numbers::e - 1};
163
164   std::vector<std::unique_ptr<IntegrationMethod>> methods;
165   methods.push_back(std::make_unique<RectangleMethod>());
166   methods.push_back(std::make_unique<TrapezoidMethod>());
167   methods.push_back(std::make_unique<SimpsonMethod>());
168
169   std::vector<Approximation> approxs;
170   approxs.reserve(methods.size());
171
172   for (const auto& method : methods) {
173     approxs.push_back(CalculateApproximation(i, *method, kEps));
174   }
175
176   std::cout << "Actual value: " << i.val << '\n';
177   PrintTable(methods, approxs);
178 }
```

4 Тестирование

В листинге 4.1 представлены результаты работы программы.

Листинг 4.1 – Результаты работы программы

```
Actual value: 1.71828
   Method  n         I              R     I + R
Rectangle  8   1.71716   0.000478341   1.71764
Trapezoid  8   1.72052  -0.000957616   1.71956
  Simpson  2   1.71832  -1.74939e-05    1.7183
```