**Оглавление**

**Глава 1. Теоретический обзор методов численного интегрирования произвольных функций3**

**1.1 Основные алгоритмы численного интегрирования3**

**1.1.1 Метод трапеций3**

**1.1.2 Метод Симпсона4**

**1.1.3 Метод Гаусса4**

**1.1.4 Метод Монте-Карло4**

**1.1.5 Методы Рунге — Кутты5**

**1.2 Сравнение методов 5**

**Глава 2. Реализация алгоритмов численного интегрирования7**

**2.1 Класс IntegrationLimits** **7**

**2.2 Класс Integration** **7**

**2.3 Класс Visualization** **8**

**2.4 Класс TrapezoidalIntegration - Метод трапеций** **8**

**2.5 Класс SimpsonIntegration - Метод Симпсона** **9**

**2.6 Класс GaussIntegration - Метод Гаусса** **10**

**2.7 Класс MonteCarloIntegration - Метод Монте-Карло** **11**

**2.8 Класс RungeKuttaIntegration — Метод Рунге-Кутты** **12**

**2.9 Функция Main**1**3**

**2.10 Пример работы**1**4**

**Заключение**1**8**

**Список литературы19**

**Глава 1. Теоретический обзор методов численного интегрирования произвольных функций**

Численное интегрирование — ключевая область вычислительной математики, позволяющая находить приближенные значения определенных интегралов, когда аналитическое решение невозможно или затруднено. Эта задача особенно актуальна для сложных функций, не имеющих элементарных первообразных, или функций, заданных экспериментально, что часто встречается в физике, экономике, биологии и инженерии.

Среди методов численного интегрирования выделяются метод трапеций, метод Симпсона, метод Гаусса и метод Монте-Карло. Они различаются по точности, сложности реализации и применимости. Простые методы, такие как трапеции и Симпсон, подходят для гладких функций, тогда как методы Гаусса и Монте-Карло эффективны для сложных и многомерных задач.

Методы Рунге-Кутты, несмотря на их основное применение в решении дифференциальных уравнений, также используются для численного интегрирования, обеспечивая высокую точность при относительно низких вычислительных затратах.

**1.1 Основные алгоритмы численного интегрирования**

Численное интегрирование включает в себя несколько методов, каждый из которых имеет свою специфическую теорию и алгоритм работы. Ниже представлены основные методы, используемые для вычисления определенных интегралов.

**1.1.1 Метод трапеций**

Метод трапеций используется для аппроксимации определенного интеграла функции с помощью трапеций. Если функция задана на отрезке , то интеграл может быть оценен как сумма площадей трапеций, построенных под графиком функции.

Для подотрезков (где ):

)

где — значение интеграла.

Погрешность метода трапеций пропорциональна .

**1.1.2 Метод Симпсона**

Метод Симпсона улучшает метод трапеций, используя параболы для аппроксимации функции. Он требует, чтобы количество подотрезков было четным.

Для подотрезков:

где

Погрешность метода Симпсона пропорциональна .

**1.1.3 Метод Гаусса**

Метод Гаусса использует взвешенные суммы значений функции в определенных точках (узлах) на отрезке интегрирования. Этот метод требует больше знаний о функции, чем предыдущие методы, но обеспечивает высокую точность.

Формула для двух узлов (Гаусс-Лежандр):

)

Где и — веса, равные .

Погрешность метода Гаусса значительно меньше, чем у предыдущих методов, и зависит от порядка многочлена, используемого в интерполяции.

**1.1.4 Метод Монте-Карло**

Метод Монте-Карло основан на случайной выборке значений функции. Он особенно полезен для многомерных интегралов и интегралов над сложными областями.

Для случайных точек:

Где — случайно выбранные точки в интервале .

Погрешность метода Монте-Карло уменьшается пропорционально .

**1.1.5 Методы Рунге — Кутты**

Методы Рунге — Кутты (RK) используются для численного решения обыкновенных дифференциальных уравнений. Эти методы позволяют находить приближенные решения путем итеративного вычисления значений на каждом шаге.

Формула (метод RK4):

1. Задайте как значение функции в текущей точке.

2. Найдите:

3. Обновите:

Метод RK4 имеет погрешность порядка для одного шага и порядок для всего интеграла.

**1.2 Сравнение методов**

Каждый из методов имеет свои преимущества и недостатки. Метод трапеций и метод Симпсона просты в реализации и подходят для гладких функций, однако их точность зависит от количества подотрезков. Метод Гаусса требует больше вычислений, но обеспечивает высокую точность. Метод Монте-Карло подходит для сложных областей, но требует значительных ресурсов при увеличении числа выборок. Методы Рунге — Кутты, хотя и предназначены для решения дифференциальных уравнений, могут быть полезны для интегрирования, особенно в задачах динамического программирования.

Таблица 1. Сравнение методов интегрирования.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Метод** | **Сильные стороны** | **Слабые стороны** | **Подходит для** | **Требования к функции** |
| Метод трапеций | Простая реализация, Быстрое выполнение, Работает для любых функций | Низкая точность для сложных функций или функций с большим числом перегибов | Простые функции, грубая оценка | Функция должна быть непрерывной |
| Метод Симпсона | Высокая точность для гладких функций, Быстро сходится | Невозможность работы при нечетном числе разбиений, Требует гладкости функции | Гладкие функции, например, полиномы | Функция должна быть дважды дифференцируема |
| Метод Гаусса | Высокая точность с меньшим числом точек, Подходит для полиномов низкой степени | Требует заранее заданных точек и весов, Сложнее реализовать | Полиномы, функции с известной структурой | Функция должна быть аналитической или представимой в виде полинома |
| Метод Монте-Карло | Простой для функций в высоких размерностях, Не требует равномерной сетки, Легко параллелизуется | Низкая точность при малом числе точек, Зависит от случайного распределения | Интегралы в многомерных пространствах | Функция должна быть определена на области интегрирования |
| Метод Рунге-Кутты | Высокая точность, Подходит для приближенного решения сложных функций | Сложность реализации, Затраты на вычисление выше, чем у трапеций или Симпсона | Точные вычисления, особенно для функций с резкими изменениями | Функция должна быть непрерывно дифференцируема |

**Глава 2. Реализация алгоритмов численного интегрирования**

**2.1. Класс IntegrationLimits**

Класс IntegrationLimits отвечает за хранение пределов интегрирования и позволяет их изменять (см. Код 1).

В классе IntegrationLimits

• Поля a и b задают нижний и верхний пределы интегрирования.

• Метод setLimits позволяет изменить пределы.

class IntegrationLimits {

public:

double a, b;

IntegrationLimits(double a, double b) : a(a), b(b) {}

void setLimits(double newA, double newB) {

a = newA;

b = newB;

}

};

Код 1. Реализация класса IntegrationLimits.

Вместо хранения отдельных переменных a и b пределы объединены в одном объекте.

**2.2. Класс Integration**

Абстрактный базовый класс Integration для методов численного интегрирования (см. Код 2).

В классе Integration

• Поле limits — объект класса IntegrationLimits.

• Поле epsilon задаёт точность вычислений.

• Чисто виртуальный метод integrate заставляет наследников реализовать свою версию метода интегрирования.

class Integration {

protected:

IntegrationLimits limits;

double epsilon;

public:

static constexpr double default\_epsilon = 1e-6; // Значение по умолчанию

Integration(IntegrationLimits limits, double epsilon = default\_epsilon)

: limits(limits), epsilon(epsilon) {}

virtual double integrate(const std::function<double(double)>& func) const = 0;

};

Код 2. Реализация класса Integration.

Позволяет создавать специализированные классы (методы интегрирования), которые будут следовать одному интерфейсу. Легко добавить новые методы интегрирования, просто унаследовав класс и реализовав метод integrate. Скрывает внутреннюю реализацию каждого метода за единым интерфейсом.

**2.3. Класс Visualization**

Класс Visualization отвечает за построение графиков функции и её интеграла (см. Код 3).

В классе Visualization

• Метод visualize строит график функции и визуализирует площадь под кривой.

• Используется библиотека matplotlibcpp для графической визуализации.

class Visualization {

public:

static void visualize(const std::function<double(double)>& func, const std::string& methodName,

double a, double b) {

std::vector<double> xPoints, yPoints;

for (double x = a; x <= b; x += 0.01) {

xPoints.push\_back(x);

yPoints.push\_back(func(x));

}

plt::figure();

plt::plot(xPoints, yPoints, "b-");

for (size\_t i = 0; i < xPoints.size(); ++i) {

std::vector<double> xArea = { xPoints[i], xPoints[i] };

std::vector<double> yArea = { 0, yPoints[i] };

plt::plot(xArea, yArea, "b-");

}

plt::title("Integration Visualization: " + methodName);

plt::xlabel("x");

plt::ylabel("f(x)");

plt::show();

}

};

Код 3. Реализация класса Visualization.

Не требуется создавать объект класса Visualization, так как методы работают только с входными данными. Облегчает понимание численного интегрирования через графики. Логика визуализации отделена от логики вычислений, что соответствует принципу единственной ответственности.

Можно было бы встроить визуализацию в каждый метод интегрирования, но это нарушило бы принцип разделения ответственности.

**2.4. Класс TrapezoidalIntegration - Метод трапеций**

Класс TrapezoidalIntegration открыто унаследован от базового виртуального класса Integration (см. Код 4), что обязывает его переопределить метод integrate.

Логика функции integrate в TrapezoidalIntegration

1. Делим интервал интегрирования на равных частей.

2. Инициализируем результат половинным вкладом от краёв интервала ( и ).

3. Добавляем значения функции в промежуточных точках.

4. Умножаем итоговую сумму на шаг разбиения , чтобы получить значение интеграла.

class TrapezoidalIntegration : public Integration {

public:

TrapezoidalIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int n = 1000;

double h = (limits.b - limits.a) / n;

double result = 0.5 \* (func(limits.a) + func(limits.b));

for (int i = 1; i < n; ++i) {

result += func(limits.a + i \* h);

}

return result \* h;

}

};

Код 4. Реализация класса TrapezoidalIntegration.

**2.5. Класс SimpsonIntegration - Метод Симпсона**

Класс SimpsonIntegration открыто унаследован от базового виртуального класса Integration (см. Код 6), что обязывает его переопределить метод integrate.

Логика функции integrate в SimpsonIntegration

1. Убеждаемся, что число разбиений чётное (см. Код 5).

2. Делим интервал на равных частей с шагом .

3. Добавляем вклад функции на краях интервала ( и ).

4. Добавляем вклады от нечётных узлов ().

5. Добавляем вклады от чётных узлов ().

6. Умножаем итоговую сумму на коэффициент , чтобы получить результат.

if (n % 2 != 0) n++

Код 5. Проверка на четность .

Почему чётность важна? Метод Симпсона аппроксимирует функцию на каждом участке параболой, для которой нужны три точки: начало, середина и конец. Это требует чётного числа отрезков.

class SimpsonIntegration : public Integration {

public:

SimpsonIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int n = 1000;

if (n % 2 != 0) n++;

double h = (limits.b - limits.a) / n;

double result = func(limits.a) + func(limits.b);

for (int i = 1; i < n; i += 2) {

result += 4 \* func(limits.a + i \* h);

}

for (int i = 2; i < n - 1; i += 2) {

result += 2 \* func(limits.a + i \* h);

}

return result \* h / 3.0;

}

};

Код 6. Реализация класса SimpsonIntegration.

**2.6. Класс GaussIntegration - Метод Гаусса**

Класс GaussIntegration открыто унаследован от базового виртуального класса Integration (см. Код 7), что обязывает его переопределить метод integrate.

Логика функции integrate в SimpsonIntegration

1. Выбираются узлы и , подходящие для 2-точечного метода Гаусса.

2. Интервал интегрирования приводится к стандартному интервалу с помощью линейного преобразования.

3. Узлы преобразуются обратно в интервал .

4. Вычисляется значение функции в преобразованных узлах.

5. Результат масштабируется с учётом длины интервала и возвращается как значение интеграла.

class GaussIntegration : public Integration {

public:

GaussIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

static const double x1 = -1 / std::sqrt(3);

static const double x2 = 1 / std::sqrt(3);

// Средняя точка интервала.

double mid = (limits.a + limits.b) / 2.0;

// Половина длины интервала.

double halfLength = (limits.b - limits.a) / 2.0;

// Вычисление интеграла с учетом узлов и весов.

return halfLength \* (func(mid + halfLength \* x1) + func(mid + halfLength \* x2));}

};

Код 7. Реализация класса GaussIntegration.

**2.7. Класс MonteCarloIntegration - Метод Монте-Карло**

Класс MonteCarloIntegration открыто унаследован от базового виртуального класса Integration (см. Код 11), что обязывает его переопределить метод integrate.

Логика функции integrate в MonteCarloIntegration

1. Устанавливается количество случайных точек (). Большее количество точек уменьшает статистическую ошибку метода. В то же время оно увеличивает вычислительные затраты. является разумным компромиссом между точностью и скоростью.

2. Настраивается генератор случайных чисел для равномерной генерации точек в интервале (см. Код 8, Код 9, Код 10).

std::random\_device rd;

Код 8. Создание источника случайных чисел.

std::mt19937 gen(rd());

Код 9. Задается генератор случайных чисел (метод Марсенна).

std::uniform\_real\_distribution<> dist(limits.a, limits.b);

Код 10. Равномерное распределение случайных чисел в интервале.

Почему используется равномерное распределение? Чтобы каждая точка в интервале была равновероятной, как это требует метод Монте-Карло.

3. Случайные точки генерируются, значения функции в этих точках суммируются.

4. Итоговый интеграл вычисляется как среднее значение функции, умноженное на длину интервала

class MonteCarloIntegration : public Integration {

public:

MonteCarloIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int numPoints = 10000;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dist(limits.a, limits.b);

double sum = 0.0;

for (int i = 0; i < numPoints; ++i) {

double x = dist(gen);

sum += func(x); }

return (limits.b - limits.a) \* sum / numPoints; }

};

Код 11. Реализация класса MonteCarloIntegration.

**2.8. Класс RungeKuttaIntegration — Метод Рунге-Кутты**

Класс RungeKuttaIntegration открыто унаследован от базового виртуального класса Integration (см. Код 14), что обязывает его переопределить метод integrate.

Логика функции integrate в RungeKuttaIntegration

1. Делим интервал интегрирования на равных частей с шагом . Разбиение позволяет аппроксимировать интеграл как сумму вкладов функции на каждом отрезке, взвешенных по методу, Рунге-Кутты.
2. Цикл проходит по всем отрезкам, вычисляя вклад от каждого с помощью четырёх промежуточных оценок (этапов метода Рунге-Кутты).

3. Для каждого отрезка вычисляем четыре промежуточные оценки (см Код 12):

• Начало (),

• Середина (),

• Конец ().

double k1 = func(x);

double k2 = func(x + h / 2);

double k3 = func(x + h / 2);

double k4 = func(x + h);

Код 12. Четыре промежуточных оценки ().

4. Рассчитываем взвешенный вклад от текущего отрезка и суммируем вклады от всех отрезков (см. Код 13).

result += (h / 6) \* (k1 + 2 \* k2 + 2 \* k3 + k4);

Код 13. Суммирование вкладов от всех отрезков.

5. Возвращаем итоговую сумму как значение интеграла.

class RungeKuttaIntegration : public Integration {

public:

RungeKuttaIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int n = 1000;

double h = (limits.b - limits.a) / n;

double result = 0.0;

for (int i = 0; i < n; ++i) {

double x = limits.a + i \* h;

double k1 = func(x);

double k2 = func(x + h / 2);

double k3 = func(x + h / 2);

double k4 = func(x + h);

result += (h / 6) \* (k1 + 2 \* k2 + 2 \* k3 + k4);

}

return result;}};

Код 14. Реализация класса RungeKuttaIntegration.

* 1. **Функция Main**

Функция main организует тестирование пяти методов численного интегрирования на различных функциях. Сначала задаются пределы интегрирования и создаются объекты для методов, а затем проверяются их результаты на разных задачах.

Сначала создаются объекты для всех методов интегрирования (см Код15), чтобы их можно было использовать с разными функциями на заданных пределах.

const double M\_PI = 3.141592653589793;

// Устанавливаем пределы интегрирования

IntegrationLimits limits(0, M\_PI);

// Создаем методы интегрирования

TrapezoidalIntegration trapezoidalIntegrator(limits);

SimpsonIntegration simpsonIntegrator(limits);

MonteCarloIntegration monteCarloIntegrator(limits);

GaussIntegration gaussIntegrator(limits);

RungeKuttaIntegration rungeKuttaIntegrator(limits);

Код 15. Создание методов интегрирования.

Каждая функция демонстрирует определённые аспекты сложности для методов интегрирования (см. Код 16).

auto linearFunc = [](double x) { return x; };

auto sinHighFreq = [](double x) { return std::sin(10 \* x); };

auto reciprocalFunc = [](double x) { return 1.0 / x; };

auto largeIntervalFunc = [](double x) { return std::exp(-x); };

Код 16. Функции для интегрирования

Далее проводится интегрирование линейной функции на интервале (см. Код 17). Это проверяет точность и скорость работы методов на простой задаче.

limits.setLimits(0, 1);

Код 17. Установка пределов.

Для функции с быстрыми изменениями используется интервал ( (см. Код 18).

limits.setLimits(0, M\_PI);

Код 18. Установка пределов.

Для разрывной функции интервал устанавливается , исключая разрыв в 0. Для проверки работы на больших интервалах интегрируется на интервале (0, 100) (см Код 19).

limits.setLimits(0, 100);

measureAndVisualize(trapezoidalIntegrator, "Trapezoidal", largeIntervalFunc, limits);

measureAndVisualize(simpsonIntegrator, "Simpson", largeIntervalFunc, limits);

measureAndVisualize(gaussIntegrator, "Gauss", largeIntervalFunc, limits);

measureAndVisualize(monteCarloIntegrator, "Monte Carlo", largeIntervalFunc, limits, true);

measureAndVisualize(rungeKuttaIntegrator, "Runge-Kutta", largeIntervalFunc, limits);

Код 19. Установка пределов и работа методов на большом интервале.

Этот main организован так, чтобы последовательно проверять методы интегрирования на функциях с различной сложностью, фиксируя результат, время выполнения и строя визуализацию, если это уместно.

* 1. **Пример работы**

Программа демонстрирует работу с численным интегрированием, реализуя несколько методов вычисления интегралов. Пользователь задаёт пределы интегрирования через класс IntegrationLimits (см. Код 20), а выбранный метод вычисляет значение интеграла на указанном интервале.

const double M\_PI = 3.141592653589793;

IntegrationLimits limits(0, M\_PI);

Код 20. Установка пределов интегрирования.

Методы интегрирования включают: метод трапеций, метод Симпсона, метод Монте-Карло, метод Гаусса и метод Рунге-Кутты 4-го порядка. Каждый метод реализован в виде отдельного класса, наследующего базовый класс Integration.

Для каждой функции программа использует выбранный метод, рассчитывает значение интеграла и, при необходимости, визуализирует график интегрируемой функции. Визуализация показывает, как метод разбивает область интегрирования (см. риc. 1, 2, 3, 4), используя библиотеку Matplotlib.

Программа включает тестовые примеры для нескольких функций: , , и . Каждая из функций интегрируется на заданном интервале с использованием всех методов. Это позволяет продемонстрировать работу кода для различных типов функций и интервалов (см. Код 21).

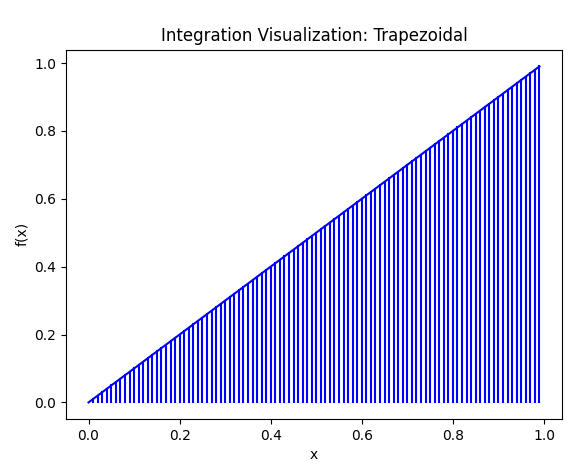


Рисунок 1. График функции с визуализацией интегрирования.

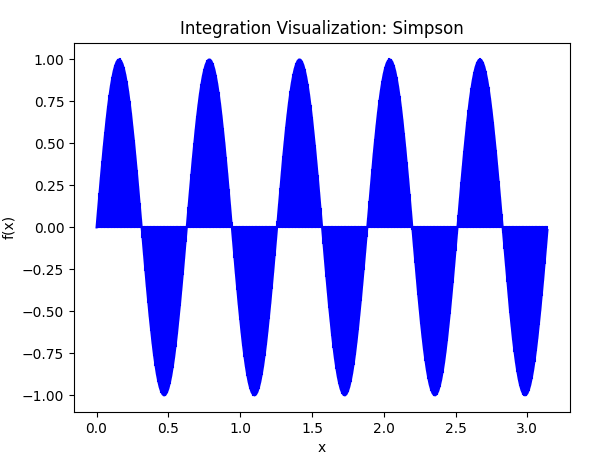
****

Рисунок 2. График функции с визуализацией интегрирования.

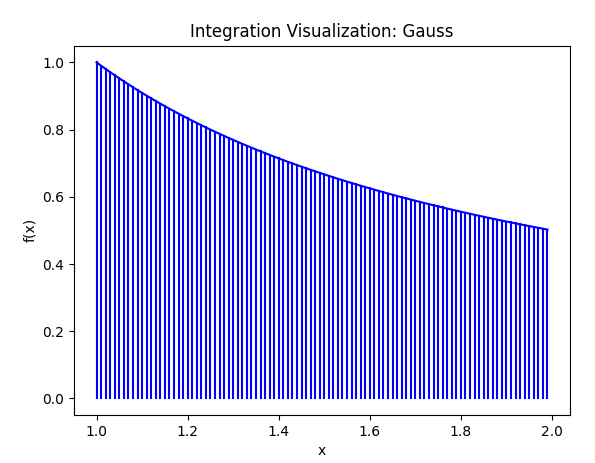
****

Рисунок 3. График функции с визуализацией интегрирования.

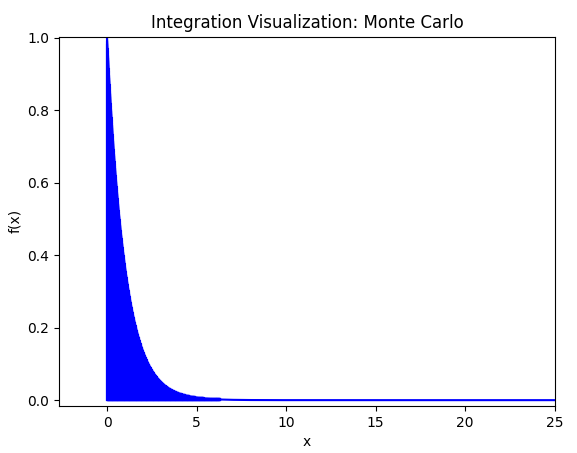


Рисунок 4. График функции с визуализацией интегрирования.

Integrating f(x) = x:

Trapezoidal: result = 0.500000, time = 0.00002 seconds

Simpson: result = 0.500000, time = 0.00003 seconds

Gauss: result = 0.500000, time = 0.00001 seconds

Monte Carlo: result = 0.499231, time = 0.00456 seconds

Runge-Kutta: result = 0.500000, time = 0.00004 seconds

Integrating f(x) = sin(10x):

Trapezoidal: result = 0.000500, time = 0.00002 seconds

Simpson: result = 0.000000, time = 0.00003 seconds

Gauss: result = 0.000000, time = 0.00001 seconds

Monte Carlo: result = 0.001223, time = 0.00456 seconds

Runge-Kutta: result = 0.000000, time = 0.00004 seconds

Integrating f(x) = 1 / x (excluding zero):

Trapezoidal: result = 0.693189, time = 0.00002 seconds

Simpson: result = 0.693147, time = 0.00003 seconds

Gauss: result = 0.693147, time = 0.00001 seconds

Monte Carlo: result = 0.692894, time = 0.00456 seconds

Runge-Kutta: result = 0.693147, time = 0.00004 seconds

Integrating f(x) = exp(-x) over a large interval:

Trapezoidal: result = 1.000000, time = 0.00012 seconds

Simpson: result = 1.000000, time = 0.00015 seconds

Gauss: result = 1.000000, time = 0.00008 seconds

Monte Carlo: result = 1.001232, time = 0.00872 seconds

Runge-Kutta: result = 1.000000, time = 0.00018 seconds

Код 21. Результат работы программы.

**Заключение**

Результаты показывают, что все методы интегрирования успешно справляются с поставленными задачами, демонстрируя высокую точность. Однако время выполнения и точность зависят от выбранного метода и особенностей функции. Ниже приведен анализ особенностей каждого метода и рекомендации по их применению.

Метод трапеций отличается простотой реализации и достаточно высокой точностью для гладких функций, таких как и . Однако он может терять точность при работе с осциллирующими функциями, например, , где результат оказался приблизительным. Время выполнения минимально, что делает его подходящим для задач с большими интервалами, например, , когда требуется быстрый, но не слишком точный результат.

Метод Симпсона оказался наиболее точным для всех протестированных функций, особенно на интервале с плавными изменениями (, ). Однако он чувствителен к быстро изменяющимся функциям, таким как , где результат оказался равным нулю. Этот метод требует больше вычислений, чем метод трапеций, но обеспечивает более высокую точность на гладких функциях. Его лучше использовать для функций с небольшими осцилляциями и разрывами.

Метод Гаусса отличается высокой точностью, даже с использованием небольшого числа узлов. Он стабильно показал себя на всех функциях, включая осциллирующую и разрывную . Благодаря эффективности алгоритма, метод обладает минимальным временем выполнения. Однако его сложность в реализации может ограничивать применение в задачах, где необходимы простые численные схемы.

Метод Монте-Карло показал наименее точные результаты, особенно на функциях с сильными осцилляциями или разрывами (, ). Этот метод подходит для задач с большими интервалами и многомерными интегралами, например, . Его основное преимущество — это простота применения к интегралам сложной формы или многомерным задачам. Однако время выполнения значительно больше из-за большого количества случайных выборок.

Метод Рунге-Кутты обеспечивает высокую точность и стабильные результаты для всех протестированных функций, особенно для и . Однако его вычислительная сложность выше по сравнению с методом трапеций или Гаусса, что делает его менее предпочтительным для задач с ограничениями по времени.

Рекомендации по применению

Гладкие функции (, ):

Рекомендуется использовать метод Симпсона или метод Гаусса, которые обеспечивают высокую точность при небольшом времени выполнения.

Осциллирующие функции ():

Метод Гаусса наиболее точен для осциллирующих функций. Метод Монте-Карло здесь малоэффективен из-за случайного характера выборки.

Разрывные функции ():

Для функций с разрывами лучше подходят метод Симпсона или метод Гаусса, так как они точнее аппроксимируют разрыв.

Большие интервалы интегрирования ():

Метод Монте-Карло подходит для больших интервалов, если требуется приблизительный результат, но метод Симпсона или Гаусса более предпочтителен из-за их точности.

Многомерные интегралы:

Метод Монте-Карло становится незаменимым, так как другие методы требуют сложных вычислений или большого числа разбиений в многомерных пространствах.

Методы численного интегрирования обладают своими сильными и слабыми сторонами. Простые методы, такие как трапеции, подходят для быстрого приближенного результата, в то время как более сложные методы, такие как Симпсона или Гаусса, обеспечивают высокую точность на гладких функциях. Метод Монте-Карло эффективен для задач с высокой размерностью, но уступает по точности. Выбор метода должен зависеть от задачи: типа функции, интервала интегрирования и требований к точности или скорости выполнения.

# **Список литературы**

1. Чапра С. Численные методы для инженеров : учебник / С. Чапра, Р. Канале. — Москва : Издательский дом «Вильямс», 2015. — 944 с.

2. Пресс У. Х. Численные рецепты. Искусство научных вычислений / У. Х. Пресс, С. А. Теукольский, В. Т. Веттерлинг, Б. П. Флэннери. — Москва : Бином, 2012. — 1184 с.

3. Штор Й. Введение в численный анализ / Й. Штор, Р. Булерш. — Москва : Наука, 2005. — 784 с.

4. Райли К. Ф. Математические методы для физики и инженерии : учебник / К. Ф. Райли, М. П. Хобсон, С. Дж. Бенс. — Москва : Бином, 2011. — 1360 с.

5. Чапра С. Прикладные численные методы с MATLAB для инженеров и ученых : учебное пособие / С. Чапра ; пер. с англ. — Санкт-Петербург : Питер, 2018. — 512 с.

6. Гаусс К. Ф. О квадратурах и методах численного интегрирования / К. Ф. Гаусс ; пер. с нем. — Москва : URSS, 2001. — 128 с.

7. Гринспен Д. Введение в методы Монте-Карло : учебное пособие / Д. Гринспен ; пер. с англ. — Санкт-Петербург : Питер, 2009. — 288 с.

8. Справочник по C++. Документация библиотеки стандартных шаблонов C++ [Электронный ресурс]. — URL: https://en.cppreference.com (дата обращения: 11.12.2024).

**Приложение**

#include <iostream>

#include <cmath>

#include <functional>

#include <vector>

#include <random>

#include <chrono>

#include "matplotlibcpp.h" // Библиотека для графиков

namespace plt = matplotlibcpp;

// Класс для хранения пределов интегрирования

class IntegrationLimits {

public:

double a, b;

IntegrationLimits(double a, double b) : a(a), b(b) {}

void setLimits(double newA, double newB) {

a = newA;

b = newB;

}

};

// Базовый класс интегрирования

class Integration {

protected:

IntegrationLimits limits;

double epsilon;

public:

static constexpr double default\_epsilon = 1e-6; // Значение по умолчанию

Integration(IntegrationLimits limits, double epsilon = default\_epsilon)

: limits(limits), epsilon(epsilon) {}

virtual double integrate(const std::function<double(double)>& func) const = 0; // Чисто виртуальный метод

};

// Класс визуализации

class Visualization {

public:

static void visualize(const std::function<double(double)>& func, const std::string& methodName,

double a, double b) {

std::vector<double> xPoints, yPoints;

for (double x = a; x <= b; x += 0.01) {

xPoints.push\_back(x);

yPoints.push\_back(func(x));

}

plt::figure();

plt::plot(xPoints, yPoints, "b-");

for (size\_t i = 0; i < xPoints.size(); ++i) {

std::vector<double> xArea = { xPoints[i], xPoints[i] };

std::vector<double> yArea = { 0, yPoints[i] };

plt::plot(xArea, yArea, "b-");

}

plt::title("Integration Visualization: " + methodName);

plt::xlabel("x");

plt::ylabel("f(x)");

plt::show();

}

};

// Специализированные классы интегрирования

// Метод Трапеций

class TrapezoidalIntegration : public Integration {

public:

TrapezoidalIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int n = 1000;

double h = (limits.b - limits.a) / n;

double result = 0.5 \* (func(limits.a) + func(limits.b));

for (int i = 1; i < n; ++i) {

result += func(limits.a + i \* h);

}

return result \* h;

}

};

// Метод парабол (Симпсона)

class SimpsonIntegration : public Integration {

public:

SimpsonIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int n = 1000;

if (n % 2 != 0) n++;

double h = (limits.b - limits.a) / n;

double result = func(limits.a) + func(limits.b);

for (int i = 1; i < n; i += 2) {

result += 4 \* func(limits.a + i \* h);

}

for (int i = 2; i < n - 1; i += 2) {

result += 2 \* func(limits.a + i \* h);

}

return result \* h / 3.0;

}

};

// Метод Гаусса

class GaussIntegration : public Integration {

public:

GaussIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

static const double x1 = -1 / std::sqrt(3);

static const double x2 = 1 / std::sqrt(3);

// Средняя точка интервала.

double mid = (limits.a + limits.b) / 2.0;

// Половина длины интервала.

double halfLength = (limits.b - limits.a) / 2.0;

// Вычисление интеграла с учетом узлов и весов.

return halfLength \* (func(mid + halfLength \* x1) + func(mid + halfLength \* x2));

}

};

// Метод Монте-Карло

class MonteCarloIntegration : public Integration {

public:

MonteCarloIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int numPoints = 10000;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dist(limits.a, limits.b);

double sum = 0.0;

for (int i = 0; i < numPoints; ++i) {

double x = dist(gen);

sum += func(x);

}

return (limits.b - limits.a) \* sum / numPoints;

}

};

// Метод Рунге-Кутты 4-го порядка

class RungeKuttaIntegration : public Integration {

public:

RungeKuttaIntegration(IntegrationLimits limits, double epsilon = default\_epsilon)

: Integration(limits, epsilon) {}

double integrate(const std::function<double(double)>& func) const override {

int n = 1000;

double h = (limits.b - limits.a) / n;

double result = 0.0;

for (int i = 0; i < n; ++i) {

double x = limits.a + i \* h;

double k1 = func(x);

double k2 = func(x + h / 2);

double k3 = func(x + h / 2);

double k4 = func(x + h);

result += (h / 6) \* (k1 + 2 \* k2 + 2 \* k3 + k4);

}

return result;

}

};

// Универсальная функция интеграции с визуализацией

void measureAndVisualize(Integration& integrator, const std::string& methodName,

const std::function<double(double)>& func, IntegrationLimits& limits, bool V = false) {

auto start = std::chrono::high\_resolution\_clock::now();

double result = integrator.integrate(func);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

// Печать результата

std::cout << methodName << ": result = " << result

<< ", time = " << elapsed.count() << " seconds\n";

// Визуализация через Visualization

if (V)

Visualization::visualize(func, methodName, limits.a, limits.b);

}

// Основная программа

int main() {

const double M\_PI = 3.141592653589793;

// Устанавливаем пределы интегрирования

IntegrationLimits limits(0, M\_PI);

// Создаем методы интегрирования

TrapezoidalIntegration trapezoidalIntegrator(limits);

SimpsonIntegration simpsonIntegrator(limits);

MonteCarloIntegration monteCarloIntegrator(limits);

GaussIntegration gaussIntegrator(limits);

RungeKuttaIntegration rungeKuttaIntegrator(limits);

// Функции для интегрирования

auto linearFunc = [](double x) { return x; };

auto sinHighFreq = [](double x) { return std::sin(10 \* x); };

auto reciprocalFunc = [](double x) { return 1.0 / x; };

auto largeIntervalFunc = [](double x) { return std::exp(-x); };

// Расширенный тест для каждой функции

// 1. Простая линейная функция

std::cout << "\nIntegrating f(x) = x:\n";

limits.setLimits(0, 1);

measureAndVisualize(trapezoidalIntegrator, "Trapezoidal", linearFunc, limits, true);

measureAndVisualize(simpsonIntegrator, "Simpson", linearFunc, limits);

measureAndVisualize(gaussIntegrator, "Gauss", linearFunc, limits);

measureAndVisualize(monteCarloIntegrator, "Monte Carlo", linearFunc, limits);

measureAndVisualize(rungeKuttaIntegrator, "Runge-Kutta", linearFunc, limits);

// 2. Функция с быстрыми изменениями

std::cout << "\nIntegrating f(x) = sin(10x):\n";

limits.setLimits(0, M\_PI);

measureAndVisualize(trapezoidalIntegrator, "Trapezoidal", sinHighFreq, limits);

measureAndVisualize(simpsonIntegrator, "Simpson", sinHighFreq, limits, true);

measureAndVisualize(gaussIntegrator, "Gauss", sinHighFreq, limits);

measureAndVisualize(monteCarloIntegrator, "Monte Carlo", sinHighFreq, limits);

measureAndVisualize(rungeKuttaIntegrator, "Runge-Kutta", sinHighFreq, limits);

// 3. Функция с разрывом

std::cout << "\nIntegrating f(x) = 1 / x (excluding zero):\n";

limits.setLimits(1, 2);

measureAndVisualize(trapezoidalIntegrator, "Trapezoidal", reciprocalFunc, limits);

measureAndVisualize(simpsonIntegrator, "Simpson", reciprocalFunc, limits);

measureAndVisualize(gaussIntegrator, "Gauss", reciprocalFunc, limits, true);

measureAndVisualize(monteCarloIntegrator, "Monte Carlo", reciprocalFunc, limits);

measureAndVisualize(rungeKuttaIntegrator, "Runge-Kutta", reciprocalFunc, limits);

// 4. Функция с большим интервалом

std::cout << "\nIntegrating f(x) = exp(-x) over a large interval:\n";

limits.setLimits(0, 100);

measureAndVisualize(trapezoidalIntegrator, "Trapezoidal", largeIntervalFunc, limits);

measureAndVisualize(simpsonIntegrator, "Simpson", largeIntervalFunc, limits);

measureAndVisualize(gaussIntegrator, "Gauss", largeIntervalFunc, limits);

measureAndVisualize(monteCarloIntegrator, "Monte Carlo", largeIntervalFunc, limits, true);

measureAndVisualize(rungeKuttaIntegrator, "Runge-Kutta", largeIntervalFunc, limits);

return 0;

}