МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

# Отчет по лабораторной работе «Вычисление многомерных интегралов методом трапеций»

**Выполнил**:
студент группы 381706-2
Синицина Мария Сергеевна

**Проверил**:
Доцент кафедры МОСТ,
кандидат технических наук
Сысоев Александр Владимирович

Нижний Новгород
2019

# Содержание

Введение....3
Постановка задачи....4
Описание алгоритма....5
Схема распараллеливания....6
Описание программной реализации....7
Подтверждение корректности....8
Эксперименты....9
Заключение....10
Литература....11
Книги:....11
Приложение....12

# Введение

Задача вычисления определенного интеграла I для некоторой заданной на отрезке [a, b] функции f(x) является классической задачей математического анализа.

В математическом анализе обосновывается аналитический способ нахождения значения интеграла с помощью формулы Ньютона- Лейбница $\int_a^b f(x)dx = F(b) - F(a)$. К сожалению, применение этого подхода к вычислению интеграла наталкивается на несколько серьезных препятствий. Во-первых, для многих элементарных функций f(x) не существует первообразной среди элементарных функций. Во-вторых, даже если первообразная F(x) для заданной функции f(x) найдена, то вычисление двух ее значений F(a) и F(b) может оказаться более трудоемким, чем вычисление существенно большего количества значений f(x). И наконец, для многих реальных приложений определенного интеграла характерна дискретность задания подынтегральной функции, что делает указанный аналитический подход не применимым в принципе. Сказанное предопределяет необходимость использования приближенных формул для вычисления определенного интеграла на основе значений подынтегральной функции f(x). Такие специальные приближенные формулы называют квадратурными формулами или формулами численного интегрирования.

Большинство методов численного интегрирования, объединены общим принципом: область интегрирования разбивается по каждой из осей на равное количество частей. В каждой из полученных маленьких областей интеграл приближается простой функцией (например, линейной), значение которой вычисляется явно (путем вычисления подынтегрального выражения в одной или нескольких точках). Ввиду линейности оператора интегрирования по областям полученные значения суммируются и представляют собой результат интегрирования.

К данному типу относятся одномерные метод прямоугольников, метод трапеций, метод парабол (метод Симпсона). Перечисленные методы обобщается и для многомерного случая.

# Постановка задачи

Основной задачей проекта является изучение метода трапеций для решения множественных интегралов различной сложности, а также разработка программы, решающей подобные определённые интегралы.

Выделим следующие подзадачи:

1. Реализация последовательного алгоритма метода трапеций для получения численного решения множественного определённого интеграла
2. Реализация параллельного алгоритма метода трапеций для получения численного решения множественного определённого интеграла. Данный алгоритм подразумевает распараллеливание задачи на заданное число процессов с целью увеличения производительности
3. Тестирование работоспособности написанных алгоритмов посредствам тестов, написанных с использованием Google C++ Testing Framework
4. Проведение анализа и сравнение времени работы последовательного и параллельного алгоритмов, осуществление оценки программы на основе произведённых экспериментов с помощью расчёта ускорения.

# Описание алгоритма

Метод трапеций — метод численного интегрирования функции одной переменной, заключающийся в замене на каждом элементарном отрезке подынтегральной функции на многочлен первой степени, то есть линейную функцию. Площадь под графиком функции аппроксимируется прямоугольными трапециями.

Пусть функция y = f(x) непрерывна на отрезке [a; b] и нам требуется вычислить определенный интеграл $\int_a^b f(x)dx$.

Разобьем отрезок [a; b] на n равных интервалов длины $h$ точками $a = x_0 < x_1 < \cdots < x_n = b$.

В этом случае шаг разбиения находим как $h = \frac{b-a}{n}$ и узлы определяем из равенства $x_i = a + i * h, i = 0, 1, \ldots, n$.

Рассмотрим подынтегральную функцию на элементарных отрезках $[x_{i-1}; x_i], i = 1,2, \ldots, n$ .

На каждом отрезке $[x_{i-1}; x_i], i = 1,2, \ldots, n$ заменим функцию *y=f(x)* отрезком прямой, проходящей через точки с координатами $(x_{i-1}; f(x_{i-1}))$ и $(x_i; f(x_i))$.

В качестве приближенного значения интеграла $\int_{x_{i-1}}^{x_i} f(x)dx$ возьмем выражение $\frac{f(x_{i-1})+f(x_i)}{2} * h$, то есть, примем $\int_{x_{i-1}}^{x_i} f(x)dx \approx \frac{f(x_{i-1})+f(x_i)}{2} * h$ .

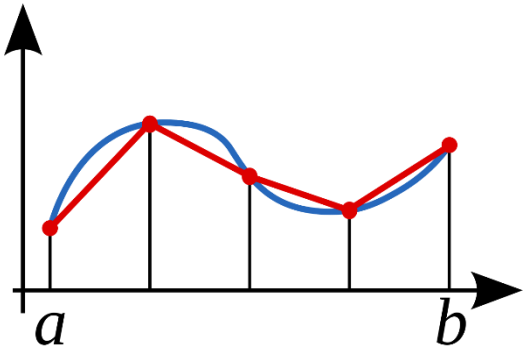


Рисунок 1. Геометрическое представление метода трапеций.

# Схема распараллеливания

В лабораторной работе осуществляется распараллеливание многомерного интеграла на заданное число процессов. В начале имеется область интегрирования, заданная отрезками [ai; bi], i = 1, …, n, количество которых совпадает с количеством интегралов **n** в многомерном интеграле. Также в примере изначально задаётся число разбиений данных отрезков; эти данные в алгоритм передаются в алгоритм виде литеры и не имеют отдельной переменной для хранения, так как для каждого примера имеется своё индивидуальное значение.

Для распараллеливания алгоритма происходит разделение данных на части и применение к ним одного и того же алгоритма. Каждый отрезок интегрирования [ai; bi], i = 1, …, n делится на заданное число разбиений, таким образом получаем величину шага для каждого отрезка. Вычисляется значения функции в каждом из таких промежутков (узлов сетки), умножается на соответствующий шаг.

# Описание программной реализации

Программа состоит из заголовочного файла sinitsina_m_trapez_integration.h, в котором объявлены функции:

double ParallelIntegr(double(*func)(std::vector<double>), std::vector <double> x, std::vector <double> y, const int n); — вычисление многомерного интеграла методом трапеций, параллельная версия

double SequentialIntegr(double(*func)(std::vector<double>), std::vector <double> x, std::vector <double> y, const int n); — вычисление многомерного интеграла методом трапеций, последовательная версия

Файла sinitsina_m_trapez_integration.cpp в котором содержится реализация данных функций и файла main.cpp в котором содержаться тесты для проверки корректности программы.

# Подтверждение корректности

Основным инструментом работоспособности программы являются тесты, разработанные при использовании Google C++ Testing Framework. Правильность получаемых результатов проверяется на четырех различных функциях разной сложности.

Тесты:

- TEST(Multiple_Trapez_Integraion, Test_1_Integral_with_2_dimension_f1) - взятие двухмерного интеграла от функции x * x + y * y по области $x \in [-5, 2]$, $y \in [10, 40]$ с числом разбиений равным 100.
- TEST(Multiple_Trapez_Integraion, Test_2_Integral_with_3_dimension_f2) - взятие трехмерного интеграла от функции (x + y + z) по области $x \in [-5,5]$, $y \in [3, 8]$, $z \in [15, 40]$ с числом разбиений равным 100.
- TEST(Multiple_Trapez_Integraion, Test_3_Integral_with_3_dimension_f3) - взятие трехмерного интеграла от функции (log10(2 * x*x) + sqrt(y) + 5 * z) по области $x \in [-5, 2]$, $y \in [18, 45]$, $z \in [15, 20]$ с числом разбиений равным 100.
- TEST(Multiple_Trapez_Integraion, Test_1_Integral_with_4_dimension_f4) - взятие четырехмерного интеграла от функции x * x + y * y(x + 5 * y - 2 * z + t) по области $x \in [-10, -5]$, $y \in [5, 8]$, $z \in [10, 14]$, $t \in [15, 20]$ с числом разбиений равным 100.

Все тесты проходят проверку, что является доказательством корректной работы программы

# Эксперименты

Эксперименты проводились на разном числе процессов для вычисления четырехмерного интеграла для функции F = x * x + y * y(x + 5 * y - 2 * z + t) по области $x \in [-10, 10]$, $y \in [5, 50]$, $z \in [0,40]$, $t \in [15, 30]$ с числом разбиений равным 100 для каждой переменной.

Вычисления проводились на ПК со следующими характеристиками:

- Процессор: AMD Ryzen 5 3500U 2.10 GHz
- Версия ОС: Windows 10 (64 бит)
- Оперативная память: 8104 МБ

| Количество процессов | Время работы последовательного алгоритма, с. | Время работы параллельного алгоритма, с. | Ускорение |
|---|---|---|---|
| 2 | 5,99497 | 3,20214 | 1,87 |
| 3 | 5,89581 | 2,56629 | 2,29 |
| 4 | 5,85127 | 1,88478 | 3,11 |
| 8 | 5,90245 | 1,62642 | 3,63 |

*Таблица 1. Время работы параллельной и последовательной версии алгоритма*

Как видно при увеличении числа процессов наблюдается значительное ускорение работы алгоритма, получается достичь почти двукратного ускорения в случае двух процессов, в случае четырех процессов мы получаем еще больший прирост ускорения работы алгоритма. На восьми процессах такого значительного прироста ускорения не происходит из-за ограничений, связанных с характеристиками процессора.

По результатам можно сделать вывод, что параллельное выполнение программы выигрывает во времени у последовательного. Это демонстрирует ускорение, формула, для вычисления которого – время работы последовательного алгоритма, делённое на время работы параллельного. Причём чем больше процессов, тем быстрее работает параллельная программа.

# Заключение

В ходе работы реализованы последовательная и параллельная версия алгоритма трапеций для вычисления многомерных интегралов. А также выполнены все поставленные подзадачи.

Проведен ряд тестов, доказывающий корректность реализованной программы. Корректность получаемых результатов проверяется на заданных примерах с помощью тестов, написанных при использовании Google C++ Testing Framework.

Проведены эксперименты, в ходе которых доказана эффективность параллельного алгоритма в сравнении с последовательным, а также показана зависимость скорости работы программы от количества запущенных процессов. С увеличением числа процессов время работы параллельной программы сильнее отличается от времени работы последовательной. Таблица 1 наглядно демонстрирует увеличение значения ускорения с увеличением числа процессов. Это обусловлено распределением данных между процессами и уменьшением объёма работы каждого процесса в отдельности, что при учёте одновременной их работы, сокращает время ожидания получения ответа при параллельных вычислениях.

# Литература

Книги:


- Самарский А.А.
  «Введение в численные методы»
  Книга написана на основе курса лекций, читавшихся автором па факультете вычислительной математики и кибернетики МГУ
  1982 год
  272 страницы

- Шарый С. П.
  «Курс Вычислительных методов»
  Институт вычислительных технологий СО РАН
  Новосибирский государственный университет
  Новосибирск – 2020
  636 страниц

- Самарский А. А., Гулин А. В.
  «Численные методы»
  Мосвка «Наука»
  Главная редакция физико-математической литературы 1989 гор
  432 страницы

- В.П. Гергель, Р.Г. Стронгин.
  Основы параллельных вычислений для многопроцессорных вычислительных систем.
  Учебное пособие.

# Приложение

### sinitsina_m_trapez_integration.h

```
// Copyright 2019 Sinitsina Maria
#ifndef MODULES_TASK_3_SINITSINA_M_TRAPEZ_INTEGRATION_SINITSINA_M_TRAPEZ_INTEGRATION_H_
#define MODULES_TASK_3_SINITSINA_M_TRAPEZ_INTEGRATION_SINITSINA_M_TRAPEZ_INTEGRATION_H_

#include <mpi.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

double ParallelIntegr(double(*func)(std::vector<double>), std::vector <double> x,
    std::vector <double> y, const int n);
double SequentialIntegr(double(*func)(std::vector<double>), std::vector <double> x,
    std::vector <double> y, const int n);

#endif  // MODULES_TASK_3_SINITSINA_M_TRAPEZ_INTEGRATION_SINITSINA_M_TRAPEZ_INTEGRATION_H_
```

### sinitsina_m_trapez_integration.cpp

```
// Copyright 2019 Sinitsina Maria
#include <vector>
#include
"../../../modules/task_3/sinitsina_m_trapez_integration/sinitsina_m_trapez_integration.h"

double ParallelIntegr(double(*func)(std::vector<double>), std::vector <double> x, std::vector
<double> y, const int n) {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (x.size() != y.size() || n <= 0) {
        throw "Wrong data";
    }
    int dimension = x.size();

    std::vector<double> h(dimension);
    std::vector<double> x1(dimension);
    std::vector<double> y1(dimension);

    int num = pow(n, dimension - 1);

    int n1 = n;
    if (rank == 0) {
        for (int i = 0; i < dimension; ++i) {
            h[i] = (y[i] - x[i]) / n;
            x1[i] = x[i];
            y1[i] = y[i];
        }
    }

    MPI_Bcast(&num, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n1, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&h[0], dimension, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&x1[0], dimension, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&y1[0], dimension, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    int ost = num % size;
    int tmp = 0;
    int number = num / size;
    if (rank < ost) {
        number += 1;
        tmp = rank * number;
    } else {
        tmp = rank * number + ost;
    }

    std::vector<std::vector<double>> segments(number);
    for (int i = 0; i < number; ++i) {
        int N = tmp + i;
        for (int j = 0; j < dimension - 1; ++j) {
            segments[i].push_back(x1[j] + h[j] * (N % n + 0.5));
        }
    }

    double preres = 0.0;
    for (int i = 0; i < number; ++i) {
        for (int j = 0; j < n; ++j) {
            segments[i].push_back(x1[dimension - 1] + (j + 0.5) * h[dimension - 1]);
            preres += func(segments[i]);
            segments[i].pop_back();
        }
    }

    double result = 0.0;
    MPI_Reduce(&preres, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    for (int i = 0; i < dimension; ++i) {
        result *= h[i];
    }
    return result;
}

double SequentialIntegr(double(*func)(std::vector<double>), std::vector <double> x,
    std::vector <double> y, const int n) {
    if (x.size() != y.size() || n <= 0) {
        throw "wrong data";
    }
    int dimension = x.size();
    std::vector<double> h(dimension);

    for (int i = 0; i < dimension; ++i) {
        h[i] = (y[i] - x[i]) / static_cast<double>(n);
    }
    std::vector <double> segments(dimension);
    double result = 0.0;
    int num = pow(n, dimension);
    for (int i = 0; i < num; ++i) {
        for (int j = 0; j < dimension; ++j) {
            segments[j] = x[j] + h[j] * (i % n + 0.5);
        }
        result += func(segments);
    }
    for (int i = 0; i < dimension; ++i) {
        result *= h[i];
    }
    return result;
}
```

**main.cpp**

```
// Copyright 2019 Sinitsina Maria
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <math.h>
#include <vector>
#include
"../../../modules/task_3/sinitsina_m_trapez_integration/sinitsina_m_trapez_integration.h"

double f1(std::vector<double> vec) {
    double x = vec[0];
    double y = vec[1];
    return (x * x + y * y);
}

double f2(std::vector<double> vec) {
    double x = vec[0];
    double y = vec[1];
    double z = vec[2];
    return (x + y + z);
}

double f3(std::vector<double> vec) {
    double x = vec[0];
    double y = vec[1];
    double z = vec[2];
    return (log10(2 * x*x) + sqrt(y) + 5 * z);
}

double f4(std::vector<double> vec) {
    double x = vec[0];
    double y = vec[1];
    double z = vec[2];
    double t = vec[3];
    return (x + 5 * y - 2 * z + t);
}

TEST(Multiple_Trapez_Integraion, Test_1_Integral_with_2_dimension_f1) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 2;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
    x = { 2, 10 };
    y = { -5, 40 };
    const int n = 100;

    double result = ParallelIntegr(f1, x, y, n);

    if (rank == 0) {
        double error = 0.0001;
        ASSERT_NEAR(result, SequentialIntegr(f1, x, y, n), error);
    }
}

TEST(Multiple_Trapez_Integraion, Test_2_Integral_with_3_dimension_f2) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 3;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
```

```
    x = { -5, 3, 15 };
    y = { 5, 8, 40 };
    const int n = 100;

    double result = ParallelIntegr(f2, x, y, n);

    if (rank == 0) {
        double error = 0.0001;
        ASSERT_NEAR(result, SequentialIntegr(f2, x, y, n), error);
    }
}

TEST(Multiple_Trapez_Integraion, Test_3_Integral_with_3_dimension_f3) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 3;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
    x = { 2, 18, 15 };
    y = { -5, 45, 20 };
    const int n = 100;

    double result = ParallelIntegr(f3, x, y, n);

    if (rank == 0) {
        double error = 0.0001;
        ASSERT_NEAR(result, SequentialIntegr(f3, x, y, n), error);
    }
}

TEST(Multiple_Trapez_Integraion, Test_4_Wrong_Data_Negative_in_n) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 2;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
    x = { 2, 1 };
    y = { -5, 6 };
    const int n = -100;

    if (rank == 0) {
        ASSERT_ANY_THROW(ParallelIntegr(f1, x, y, n));
    }
}

TEST(Multiple_Trapez_Integraion, Test_5_Wrong_Data_Negative_in_size_of_first_coord) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 2;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
    x = { 2 };
    y = { -5, 6 };
    const int n = 100;

    if (rank == 0) {
        ASSERT_ANY_THROW(ParallelIntegr(f1, x, y, n));
    }
}

TEST(Multiple_Trapez_Integraion, Test_6_Wrong_Data_Negative_in_size_of_second_coord) {
```

```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 2;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
    x = { 2, 10 };
    y = { -5, 6, 13 };
    const int n = 100;

    if (rank == 0) {
        ASSERT_ANY_THROW(ParallelIntegr(f1, x, y, n));
    }
}

TEST(Multiple_Trapez_Integraion, Test_1_Integral_with_4_dimension_f4) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dimension = 4;

    std::vector<double> x(dimension);
    std::vector<double> y(dimension);
    x = { -10, 5, 0, 15 };
    y = { 10, 50, 40, 30 };
    const int n = 100;
    double l1, l2, dl;
    l1 = MPI_Wtime();
    double result = ParallelIntegr(f4, x, y, n);
    l2 = MPI_Wtime();
    dl = l2 - l1;
    double t1, t2, dt;
    if (rank == 0) {
        double error = 0.001;
        t1 = MPI_Wtime();
        ASSERT_NEAR(result, SequentialIntegr(f4, x, y, n), error);
        t2 = MPI_Wtime();
        dt = t2 - t1;
        std::cout << "seq time  " << dt << std::endl;
        std::cout << "par time  " << dl << std::endl;

    }
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```