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 Монте-Карло»

Выполнил:
студент группы 381706-2
Крюков Дмитрий Алексеевич

Проверил:
Доцент, кандидат технических наук
Сысоев Александр Владимирович

Нижний Новгород
2019

Содержание

Введение

При переходе от задачи вычисления одномерных интегралов к многомерному интегрированию возникает целый ряд новых проблем. В одномерном случае мы имеем три возможных типа области интегрирования: конечный, полубес- конечный и бесконечный интервалы. В многомерном случае мы сталкиваемся с большим разнообразием областей интегрирования. В дополнение, подинтегральная функция может иметь особенность не только в точке, а также во множествах точек, таких как, например, интервал, плоскость и т.д. Поэтому вычисление многомерных интегралов представляет собой значительно более сложную задачу по сравнению с одномерным случаем

В небольших размерностях можно применять квадратурные формулы, основанные на многочленах Лагранжа. Однако в больших размерностях эти методы становятся неприемлемыми из-за быстрого возрастания числа точек сетки и/или сложной границы области. В этом случае применяется метод Монте-Карло. Генерируются случайные точки в нашей области и усредняются значения функции в них.

Постановка задачи

Реализовать последовательную версию алгоритма многомерного интегрирования методом Монте-Карло. Используя средства MPI реализовать параллельную версию алгоритма. Сравнить эффективность последовательной и параллельной версии и объяснить результаты.

Описание алгоритма

Интеграл оценивается как произведение площади (объема) области и среднего значения функции, которое опять вычисляется по выборке на множестве случайных точек.

Введем некоторые величины, которые позволят нам формализовать алгоритм численного интегрирования. Пусть дан двумерный интеграл:

$$\iint_{\Omega} f(x,y)dxdy$$

Таким образом, граница области $\partial\Omega$ задана неявной функцией (кривой) $g(x,y)=0$. Такое описание областей распространено в последние десятилетия, при этом g называется функцией уровня, а граница $g=0$ — нулевым контуром функции уровня. Для простых областей можно легко построить функцию g вручную, но в более сложных промышленных приложениях следует обратиться к математическим моделям построения g.

Пусть $A(\Omega)$ — площадь области Ω. Мы можем численно найти интеграл по следующему алгоритму:

1. Помещаем область Ω внутрь прямоугольника R;
2. Генерируем большое число случайных точек на R;
3. Вычисляем долю q точек, которые попали в область Ω;
4. Приближаем $A(\Omega)/A(R)$ числом q, т.е., полагаем $A(\Omega)=qA(R)$;
5. Вычисляем среднее значение $\bar{f}$ функции f на области Ω;
6. Вычисляем приближенное значение интеграла как $A(\Omega)\bar{f}$.

Отметим, что площадь A(R) прямоугольника R легко вычислить, при том что площадь $A(\Omega)$ нам не известна. Однако, если предположить, что доля площади A(R) занимаемой областью Ω такая же как доля случайных точек, попавших внутрь Ω, можно получить простое приближение для $A(\Omega)$.

Схема распараллеливания

В параллельной реализации при генерации точек каждым процессом мы получаем одни и те же числа в каждом процессе – неправильное использование генератора в параллельных вычислениях. Требуется, чтобы процессы работали с одной последовательностью случайных чисел, «выбирая» из нее нужные для обработки.

Нулевой процесс генерирует случайные точке в заданного прямоугольника и распределяет их по всем процессам. Каждый процесс вычисляет долю точек попавших в область интегрирования и среднее значение функции в этих точках, а затем отправляет эти данные в нулевой процесс. В нулевом процесс вычисляем площадь фигуры $A(\Omega)$ и среднее знамение функции $\bar{f}$, приближенное значение интеграла находим как $A(\Omega)\bar{f}$.

Описание программной реализации

Программа состоит из заголовочного файла `multidimensional_monte_karlo.h` в котором объявлены функции `multidimensionalIntegration(std::vector<double> start_point, double side, double(*pfunc)(std::vector<double>), bool(*parea)(std::vector<double>), unsigned int dimension, int point_count)` – вычисление многомерного интеграла методом Монте-Карло, параллельная версия

`multidimensionalIntegrationSequential(std::vector<double> start_point, double side, double(*pfunc)(std::vector<double>), bool(*parea)(std::vector<double>), unsigned int dimension, int point_count)` — вычисление многомерного интеграла методом Монте-Карло, последовательная версия

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

Подтверждение корректности

Для подтверждения корректности данной программы были написаны несколько тестов с использованием библиотеки. Все тесты находят значение интеграла при помощи метода Монте-Карло, а потом сравнивают их со значениями полученными аналитически.

Тесты:

- `TEST(Multidimensional_Monte_Karlo_MPI, const_func_integration_on_area_circle)` – взятие двумерного интеграла от константной функции по области круга (Вычисление площади круга при помощи двумерного интеграла)
- `TEST(Multidimensional_Monte_Karlo_MPI, linear_func_integration_on_area_2_parabols)` – взятие двумерного интеграла от функции x + y по области являющей собой пересечение двух парабол: y = x*x – 1 и y = -x*x + 1
- `TEST(Multidimensional_Monte_Karlo_MPI, const_func_integration_on_area_sphere)` -взятие трехмерного интеграла от константной функции по области сферы (Вычисление площади сферы при помощи трехмерного интеграла)
- `TEST(Multidimensional_Monte_Karlo_MPI, quadratic_func_integration_on_area_sphere)` - взятие трехмерного интеграла от функции x*x по области сферы
- `TEST(Multidimensional_Monte_Karlo_MPI, const_func_integration_on_area_three_sphere)` - четырехмерного интеграла от константной функции по области 3-сферы (Вычисление площади 3-сферы при помощи четрехмерного интеграла)

Эксперименты

Эксперименты проводились на разном числе процессов для вычисление площади 3-сферы при помощи четрехмерного интеграла, количество точек было взято равным 1 000 000.

Вычисления проводились на ПК со следующими характеристиками:

- Процессор: Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz
- Версия ОС: Windows 10 (64 бит)
- Оперативная память: 8104 МБ

Количество процессов	Время работы последовательного алгоритма	Время работы параллельного алгоритма	Ускорение
1	11.9646	12.2109	0,98
2	11.6591	7.88923	1,48
4	11.5728	6.30908	1,82
8	11.771	6.2724	1.87

Таблица 1. Время работы параллельной и последовательной версии алгоритма

Как видно при увеличении числа процессов наблюдается значительное ускорение работы алгоритма, однако накладные расходы и латентность алгоритма, связанная с необходимостью генерировать набор псевдослучайных чисел в одном процессе, не позволяет достичь двукратного ускорения в случае двух процессов, в случае четырех процессов мы получаем меньший прирост ускорения работы алгоритма, это связано с возросшими накладными расходами, а так же необходимостью использовать вычислительные конвейеры наиболее загруженные системными командами. На восьми процессах прироста ускорения не происходит из-за ограничений связанных с характеристиками процессора.

Заключение

В ходе работы реализованы последовательная и параллельная версия алгоритма Монте-Карло для вычисления многомерных интегралов. Проведен ряд тестов доказывающий корректность реализованной программы. Проведены эксперименты в ходе которых доказана эффективность параллельного алгоритма в сравнении с последовательным.

Литература

Книги:

- Соболь И.М. Численные методы Монте-Карло. – М.:Наука, 1973. – 312с.

Интернет -ресурсы:

- Метод Монте-Карло для вычисления двойного интеграла URL: https://studfile.net/preview/7373006/page:7/

Приложение

multidimensional_monte_karlo.h

```
// Copyright 2019 Kriukov Dmitry
#ifndef
MODULES_TASK_3_KRIUKOV_D_MULTIDIMENSIONAL_MONTE_KARLO_MULTIDIMENSIONAL_MONTE_KARLO_H_
#define
MODULES_TASK_3_KRIUKOV_D_MULTIDIMENSIONAL_MONTE_KARLO_MULTIDIMENSIONAL_MONTE_KARLO_H_

#include<vector>

double multidimensionalIntegration(std::vector<double> start_point, double side,
double(*pfunc)(std::vector<double>),
                                   bool(*parea)(std::vector<double>), unsigned int
dimension, int point_count);
double multidimensionalIntegrationSequential(std::vector<double> start_point, double side,
    double(*pfunc)(std::vector<double>), bool(*parea)(std::vector<double>), unsigned int
dimension, int point_count);

#endif  //
MODULES_TASK_3_KRIUKOV_D_MULTIDIMENSIONAL_MONTE_KARLO_MULTIDIMENSIONAL_MONTE_KARLO_H_
```

multidimensional_monte_karlo.cpp

```
// Copyright 2019 Kriukov Dmitry
#include <mpi.h>
#include <random>
#include <ctime>
#include <algorithm>
#include <vector>
#include <iostream>
#include "../../../modules/task_3/kriukov_d_multidimensional_monte_karlo/
multidimensional_monte_karlo.h"

double multidimensionalIntegrationSequential(std::vector<double> start_point, double side,
    double(*pfunc)(std::vector<double>), bool(*parea)(std::vector<double>), unsigned int
dimension, int point_count) {
    std::mt19937 gen;
    gen.seed(static_cast<unsigned int>(time(0)));
    std::uniform_real_distribution<> urd(0, 1);

    std::vector<double> points(dimension * point_count);
```

```
    for (unsigned int i = 0; i < points.size(); i++) {
        points[i] = start_point[i%dimension] + urd(gen)*side;
    }

    int num_inside = 0;
    double mean = 0;

    for (int i = 0; i < point_count; ++i) {
        std::vector<double> cpoint;
        for (unsigned int j = 0; j < dimension; ++j) {
            cpoint.push_back(points[dimension * i + j]);
        }
        if (parea(cpoint)) {
            num_inside++;
            mean += pfunc(cpoint);
        }
    }

    mean = mean / num_inside;
    double area = std::pow(side, dimension) * num_inside / point_count;

    return area * mean;
}


double multidimensionalIntegration(std::vector<double> start_point, double side,
double(*pfunc)(std::vector<double>),
                                   bool(*parea)(std::vector<double>), unsigned int
dimension, int point_count) {
    if (start_point.size() != dimension)
        throw(1);

    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int delta = point_count / size;
    std::vector<double> points(dimension * point_count);

    if (rank == 0) {
        std::mt19937 gen;
        gen.seed(static_cast<unsigned int>(time(NULL)));
```

```
        std::uniform_real_distribution<> urd(0, 1);
        for (unsigned int i = 0; i < points.size(); i++) {
            points[i] = start_point[i%dimension] + urd(gen)*side;
        }
    }

    std::vector<double> local_points(delta * dimension);

    MPI_Scatter(&points[0], delta * dimension, MPI_DOUBLE, &local_points[0],
                delta * dimension, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    int local_num_inside = 0;
    int num_inside = 0;
    double local_mean = 0;
    double mean = 0;

    for (int i = 0; i < delta; ++i) {
        std::vector<double> cpoint;
        for (unsigned int j = 0; j < dimension; ++j) {
            cpoint.push_back(local_points[dimension * i + j]);
        }
        if (parea(cpoint)) {
            local_num_inside++;
            local_mean += pfunc(cpoint);
        }
    }
    MPI_Reduce(&local_num_inside, &num_inside, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&local_mean, &mean, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    mean = mean / num_inside;
    double area = std::pow(side, dimension) * num_inside / point_count;

    return area * mean;
}
```

main.cpp

```
// // Copyright 2019 Kriukov Dmitry
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <math.h>
#include <vector>
#include "./multidimensional_monte_karlo.h"
```

```
#define POINT_COUNT 10000

bool area_circle_radius_2(std::vector<double> args) {
    return (args[0]* args[0] + args[1] * args[1]) < 4;
}

bool area_sphere_radius_2(std::vector<double> args) {
    return (args[0] * args[0] + args[1] * args[1] + args[2] * args[2]) < 4;
}

bool area_three_sphere_radius_2(std::vector<double> args) {
    return (args[0] * args[0] + args[1] * args[1] + args[2] * args[2] + args[3]*args[3]) <
4;
}

bool area_quart_sphere_radius_2(std::vector<double> args) {
    return (args[0] > 0 && args[2] > 0 && (args[0] * args[0] + args[1] * args[1] + args[2] *
args[2]) < 4);
}


double func_const(std::vector<double> args) {
    return 1;
}

double func_quadratic(std::vector<double> args) {
    return args[0] * args[0];
}

double func_linear(std::vector<double> args) {
    return args[0] + args[1];
}

bool area_2_parabols(std::vector<double> args) {
    return (args[1] > (args[0] * args[0] - 1)) && (args[1] < -args[0] * args[0] + 1);
}

TEST(Multidimensional_Monte_Karlo_MPI, const_func_integration_on_area_circle) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    const double abs_errror = 1.0;

    std::vector<double> s_point = { -2, -2 };

    double integration_result = multidimensionalIntegration(s_point, 4, func_const,
                                                            area_circle_radius_2, 2,
POINT_COUNT);
    if (rank == 0) {
        ASSERT_NEAR(3.14 * 4, integration_result, abs_errror);
    }
}

TEST(Multidimensional_Monte_Karlo_MPI, linear_func_integration_on_area_2_parabols) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const double abs_errror = 1.0;

    std::vector<double> s_point = { -1, -1 };

    double integration_result = multidimensionalIntegration(s_point, 2, func_linear,
area_2_parabols, 2, POINT_COUNT);

    if (rank == 0) {
        ASSERT_NEAR(0, integration_result, abs_errror);
    }
}

TEST(Multidimensional_Monte_Karlo_MPI, const_func_integration_on_area_sphere) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const double abs_errror = 5.0;

    std::vector<double> s_point = { -2, -2, -2};

    double integration_result = multidimensionalIntegration(s_point, 4, func_const,
        area_sphere_radius_2, 3, POINT_COUNT);
    if (rank == 0) {
        ASSERT_NEAR(3.14 * 4 * 8 / 3, integration_result, abs_errror);
    }
}
```

```
TEST(Multidimensional_Monte_Karlo_MPI, quadratic_func_integration_on_area_sphere) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const double abs_errror = 5.0;

    std::vector<double> s_point = { -2, -2, -2 };

    double integration_result = multidimensionalIntegration(s_point, 4, func_quadratic,
        area_quart_sphere_radius_2, 3, POINT_COUNT);
    if (rank == 0) {
        ASSERT_NEAR(3.14 * 32 / 15, integration_result, abs_errror);
    }
}

TEST(Multidimensional_Monte_Karlo_MPI, const_func_integration_on_area_three_sphere) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const double abs_errror = 10.0;

    std::vector<double> s_point = { -2, -2 , -2, -2};

    double integration_result = multidimensionalIntegration(s_point, 4, func_const,
        area_three_sphere_radius_2, 4, POINT_COUNT);
    if (rank == 0) {
        ASSERT_NEAR(3.14 * 3.14 * 16 / 2, integration_result, abs_errror);
    }
}
#define MULTIDIMENSIONAL_MONTE_KARLO_TIME_TEST

#ifdef MULTIDIMENSIONAL_MONTE_KARLO_TIME_TEST

TEST(Multidimensional_Monte_Karlo_MPI, Time_Test) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const double abs_errror = 0.5;
    const int point_count = 1000000;

    std::vector<double> s_point = { -2, -2 , -2, -2 };
```

```
    double a1, b1, a2, b2;
    double resSequaintal;

    if (rank == 0) {
        a2 = MPI_Wtime();
        resSequaintal = multidimensionalIntegrationSequential(s_point, 4, func_const,
            area_three_sphere_radius_2, 4, point_count);
        b2 = MPI_Wtime();
    }

    if (rank == 0)
        a1 = MPI_Wtime();
    double integration_result = multidimensionalIntegration(s_point, 4, func_const,
        area_three_sphere_radius_2, 4, point_count);
    if (rank == 0)
        b1 = MPI_Wtime();

    if (rank == 0) {
        std::cout << "Sequential " << b2 - a2 << std::endl;
        std::cout << "Parralel " << b1 - a1;
    }
    if (rank == 0) {
        ASSERT_NEAR(3.14 * 3.14 * 16 / 2, integration_result, abs_errror);
    }
}

#endif

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();
}
```