Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

«Вычисление многомерных интегралов методом Монте-Карло»

Выполнил:
студент группы 381706-1
Власов А. С.

Проверил:
Доцент кафедры МОСТ,
кандидат технических наук
Сысоев А. В.

Нижний Новгород
2019

Содержание

1. Введение

Методами Монте-Карло называют численные методы решения математических задач при помощи моделирования случайных величин. Однако, решать методами Монте-Карло можно любые математические задачи, а не только задачи вероятностного происхождения, связанные со случайными величинами.

До появления ЭВМ методы Монте-Карло не могли стать универсальными численными методами, ибо моделирование случайных величин вручную весьма трудоемкий процесс. Развитию методов Монте-Карло способствовало бурное развитие ЭВМ. Алгоритмы Монте-Карло сравнительно легко программируются и позволяют производить расчеты во многих задачах, недоступных для классических численных методов.

Важнейшим приемом построения методов Монте-Карло является сведение задачи к расчету математических ожиданий. Так как математические ожидания чаще всего представляют собой обычные интегралы, то центральное положение в теории метода Монте-Карло занимают методы вычисления интегралов.

2. Постановка задачи

Для выполнения цели работы были поставлены следующие задачи:

1. Реализация последовательного алгоритма вычисления многомерных интегралов методом Монте-Карло
2. Реализация параллельного алгоритма вычисления многомерных интегралов методом Монте-Карло с использованием средств MPI.
3. Проведение вычислительных экспериментов.
4. Сравнение времени работы полученных алгоритмов.

3. Метод решения

Пусть функция y = **f**(x_1, ..., x_m) непрерывна в ограниченной замкнутой области *G* и требуется вычислить *m*-кратный интеграл *I* по области *G*:

$$I = \int_G \cdots \int f(x_1, \ldots, x_m) dx_1 \ldots dx_m.$$

Геометрически число I представляет собой (m + 1)-мерный объем вертикального цилиндрического тела в пространстве $Ox_1x_2 \ldots x_my$, построенного на основании G и ограниченного сверху данной поверхностью y = f(x), где x = (x_1,…,x_m):

Преобразуем интеграл так, чтобы новая область интегрирования целиком содержалась внутри единичного m-мерного куба. Пусть область интегрирования G расположена в m-мерном параллелепипеде $a_k \le x_k \le b_k$ (k = 1,…,m).

Сделаем замену переменных: $x_k = a_k + (b_k - a_k)\xi_k$ (k = 1,…,m).

Тогда m-мерный параллелепипед преобразуется в m-мерный единичный куб

$0 \le \xi_k \le 1$, (k = 1,…,m), следовательно, новая область интегрирования **Ω** будет целиком расположена внутри этого единичного куба.

Вычислим якобиан преобразования:

$$\frac{D(x_1, \ldots, x_m)}{D(\xi_1, \ldots, \xi_m)} = \begin{vmatrix} b_1 - a_1 & 0 & \ldots & 0 \\ 0 & b_2 - a_2 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & b_m - a_m \end{vmatrix} = (b_1 - a_1) \cdot (b_2 - a_2) \cdot \ldots \cdot (b_m - a_m).$$

Таким образом,

$$I = (b_1 - a_1) \cdot (b_2 - a_2) \cdot \ldots \cdot (b_m - a_m) \cdot J$$

где

$$J = \int_\Omega \cdots \int f[a_1 + (b_1 - a_1)\xi_1, \ldots, a_m + (b_m - a_m)\xi_m] d\xi_1 \ldots d\xi_m$$

И область интегрирования **Ω** содержится внутри *m*-мерного куба.

Интеграл *J* можно оценить следующей формулой:

$$J = \text{Fcp} \cdot \Omega$$

Где Fcp = $\frac{1}{n} \sum_{i=1}^{n} F(Mi)$, Mi - случайные точки (ξ_1,…,ξ_k), а **Ω** - объем области интегрирования **Ω**, принимается равной $\frac{n}{N}$.

Таким образом имеем оценку искомого интеграла *I*:

$$I = \text{V} \cdot \text{Fcp} \cdot \Omega$$

где $V = (b_1 - a_1) \cdot (b_2 - a_2) \cdot \ldots \cdot (b_m - a_m)$ - объем параллелепипеда, ограничивающего область интегрирования G.

Отсюда получаем конечную формулу:

$$I = \frac{V}{N} \cdot \sum_{i=1}^{N} F(Mi)$$

4. Схема распараллеливания

Нулевой процесс генерирует случайные точке в заданной области интегрирования и распределяет их по всем процессам. Каждый процесс вычисляет сумму значений функции в этих точках, а затем отправляет эти данные в нулевой процесс. В нулевом процесс складываем эти суммы значений, вычисляем среднее значение и умножаем на объем параллелепипеда, ограничивающего исходную область интегрирования, получаем приближенное значение интеграла.

5. Описание программной реализации

Последовательный алгоритм вычисления многомерных интегралов методом Монте-Карло представлен функцией:

`double getIntegralMonteCarloSequential(double(*f)(std::vector<double>), const std::vector<double>& a, const std::vector<double>& b, int n)` , где `double(*f)(std::vector<double>` – подынтегральная функция, `const std::vector<double>& a, const std::vector<double>& b` – векторы нижних и верхних границ интегрирования, а `int n` – число испытаний.

Параллельный алгоритм вычисления многомерных интегралов методом Монте-Карло представлен функцией:

`double getIntegralMonteCarloParallel(double(*f)(std::vector<double>), const std::vector<double>& a, const std::vector<double>& b, int n)` с такими же аргументами, что и в функции последовательного алгоритма.

6. Подтверждение корректности

Для подтверждения корректности в программе представлен набор тестов, разработанных с помощью использования Google C++ Testing Framework.

Первый тест проверяет возникновения исключения при вызове функции с неположительным числом испытаний. А четыре других теста находят значения интегралов с разной многомерностью при помощи последовательного и параллельного алгоритма, сравнивают их между собой, а также со значениями, полученными аналитически.

Успешное прохождение всех тестов доказывает корректность работы программы.

7. Результаты экспериментов

Эксперименты проводились на ПК с следующими параметрами:

1. Операционная система: Windows 10 Домашняя
2. Процессор: Intel(R) Core(TM) i3-8130U CPU @ 2.20GHz
3. Версия Visual Studio: 2019

В рамках первого эксперимента мы берем двумерный интеграл из третьего теста.

Число испытаний	Последовательный алгоритм	Параллельный алгоритм			
		2 процесса		4 процесса	
		время, с	ускорение	время, с	ускорение
100000	0.1278	0.0913	1.399	0.0769	1.662
1000000	1.3037	0.9389	1,3885	0.8042	1.621
5000000	6.5158	4.7552	1.3702	4.0869	1.594

Таблица 1. Время работы параллельной и последовательной версии алгоритма

В рамках второго эксперимента мы берем четырехмерный интеграл из пятого теста.

Число испытаний	Последовательный алгоритм	Параллельный алгоритм			
		2 процесса		4 процесса	
		время, с	ускорение	время, с	ускорение
100000	0.1776	0.1344	1.321	0.1213	1.464
1000000	1.6859	1.2891	1,3078	1.2025	1.402
5000000	8.5681	6.6457	1.2892	6.1861	1.385

Таблица 2. Время работы параллельной и последовательной версии алгоритма

По данным, полученным в результате экспериментов, можно сделать вывод о том, что параллельный алгоритм работает действительно быстрее, чем последовательный. Однако из-за того, что нам необходимо генерировать набор псевдослучайных чисел в одном процессе, а потом передавать его другим процессам, не позволяет достичь большого ускорения. Также можно заметить, что чем более многомерный интеграл и чем больше число испытаний, тем меньше ускорение. Это объясняется в обоих случаях увеличением числа необходимых генерируемых псевдослучайных чисел.

8. Заключение

В результате лабораторной работы была реализована последовательная и параллельная версия алгоритма Монте-Карло для вычисления многомерных интегралов.

Основной задачей данной лабораторной работы была реализация параллельной версии алгоритма. Эта задача была успешно достигнута, о чем говорят результаты экспериментов, проведенных в ходе работы. Они показывают, что параллельный случай работает действительно быстрее, чем последовательный.

Кроме того, были разработаны и доведены до успешного выполнения тесты, созданные для данного программного проекта с использованием Google C++ Testing Framework и необходимые для подтверждения корректности работы программы.

9. Список литературы

1. Соболь И.М. Численные методы Монте-Карло: Изд-во «Наука», 1973.
2. Терзи М.П. Бакалаврская работа «Численное интегрирование с использованием метода Монте-Карло»: ИвГУ, 2010.
3. Википедия: свободная электронная энциклопедия: на русском языке [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Метод_Монте-Карло

10. Приложение

multi_integration_monte_carlo.h

```
// Copyright 2019 Vlasov Andrey
#ifndef
MODULES_TASK_2_VLASOV_A_MULTI_INTEGRATION_MONTE_CARLO_MULTI_INTEGRATION_MONTE_CARLO_H_
#define
MODULES_TASK_2_VLASOV_A_MULTI_INTEGRATION_MONTE_CARLO_MULTI_INTEGRATION_MONTE_CARLO_H_

#include <mpi.h>
#include <vector>

double getIntegralMonteCarloSequential(double(*f)(std::vector<double>), const
std::vector<double>& a,
  const std::vector<double>& b, int n);

double getIntegralMonteCarloParallel(double(*f)(std::vector<double>), const
std::vector<double>& a,
  const std::vector<double>& b, int n);

#endif  //
MODULES_TASK_2_VLASOV_A_MULTI_INTEGRATION_MONTE_CARLO_MULTI_INTEGRATION_MONTE_CARLO_H_
```

multi_integration_monte_carlo.cpp

```
// Copyright 2019 Vlasov Andrey
#include <mpi.h>
#include <random>
#include <vector>
#include <iostream>
#include <ctime>
#include
"../../../modules/task_3/vlasov_a_multi_integration_monte_carlo/multi_integration_monte_c
arlo.h"

double getIntegralMonteCarloSequential(double(*f)(std::vector<double>), const
std::vector<double>& a, const std::vector<double>& b, int n) {
  if (n <= 0)
    throw "n is negative";
  double res = 0.0;
  std::mt19937 gen;
  gen.seed(static_cast<unsigned int>(time(0)));
  int multiplicity = a.size();
  double S = 1;
  for (int i = 0; i < multiplicity; i++)
    S *= (b[i] - a[i]);
  std::vector<std::uniform_real_distribution<double>> r(multiplicity);
  std::vector<double> r1(multiplicity);
  for (int i = 0; i < multiplicity; i++)
    r[i] = std::uniform_real_distribution<double>(a[i], b[i]);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < multiplicity; j++)
      r1[j] = r[j](gen);
    res += f(r1);
  }
  res *= S / n;
  return res;
}
```

```
double getIntegralMonteCarloParallel(double(*f)(std::vector<double>), const
std::vector<double>& a, const std::vector<double>& b, int n) {
  if (n <= 0)
    throw "n is negative";
  int rank, size;
  double local_res = 0.0, res = 0.0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  std::mt19937 gen;
  gen.seed(static_cast<unsigned int>(time(0)));
  int multiplicity = static_cast<int>(a.size());
  std::vector<double> r(static_cast<unsigned int>(n * multiplicity));
  std::vector<std::uniform_real_distribution<double>> r1(multiplicity);
  if (rank == 0) {
    for (int i = 0; i < multiplicity; i++)
      r1[i] = std::uniform_real_distribution<double>(a[i], b[i]);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < multiplicity; j++)
        r[(i * multiplicity) + j] = r1[j](gen);
  }
  int local_n = n / size;
  std::vector<double> local_r(local_n * multiplicity);
  MPI_Scatter(&r[0], local_n * multiplicity, MPI_DOUBLE, &local_r[0], local_n *
multiplicity, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  std::vector<double> r_l(multiplicity);
  for (int i = 0; i < local_n; i ++) {
    for (int j = 0; j < multiplicity; j++)
      r_l[j] = local_r[(i * multiplicity) + j];
    local_res += f(r_l);
  }
  if ((rank == 0) & (n % size != 0)) {
    for (int i = local_n * size; i < n; i++) {
      for (int j = 0; j < multiplicity; j++)
        r_l[j] = r[(i * multiplicity) + j];
      local_res += f(r_l);
    }
  }
  MPI_Reduce(&local_res, &res, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank == 0) {
    for (int i = 0; i < multiplicity; i++)
      res *= (b[i] - a[i]);
    res /= n;
  }
  return res;
}
```

main.cpp

```
// Copyright 2019 Vlasov Andrey
#include <gtest/gtest.h>
#include <gtest-mpi-listener.hpp>
#include <mpi.h>
#include <vector>
#include <cmath>
#include <ctime>
#include <iostream>
#include "./multi_integration_monte_carlo.h"

double f1(std::vector<double> x) {
  return x[0] * x[0];
}

double f2(std::vector<double> x) {
  return 3 * x[0] * x[0] * x[0] + 2 * x[1] * x[1];
```

```
}


double f3(std::vector<double> x) {
  return sin(x[0]) + 2 * x[1] + x[2] * x[2];
}

double f4(std::vector<double> x) {
  return x[0] * x[0] + 2 * x[1] - cos(x[2]) + x[3] * x[3];
}

TEST(multi_integration_monte_carlo, test1_n_is_negative) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  double a = 0.0, b = 3.0;
  if (rank == 0)
    ASSERT_ANY_THROW(getIntegralMonteCarloSequential(f1, { a }, { b }, -1000));
}

TEST(multi_integration_monte_carlo, test2_multiplicity_is_1) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  double a = 0.0, b = 3.0;
  double res = getIntegralMonteCarloParallel(f1, { a }, { b }, 1000000);
  if (rank == 0) {
    double res1 = getIntegralMonteCarloSequential(f1, { a }, { b }, 1000000);
    ASSERT_NEAR(res1, res, 0.05);
    ASSERT_NEAR(9.0, res, 0.05);
  }
}

TEST(multi_integration_monte_carlo, test3_multiplicity_is_2) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  std::vector<double>a = { 0.0, 2.5 };
  std::vector<double>b = { 1.534, 3.12 };
  double res = getIntegralMonteCarloParallel(f2, a, b, 1000000);
  if (rank == 0) {
    double res1 = getIntegralMonteCarloSequential(f2, a, b, 1000000);
    ASSERT_NEAR(res, res1, 0.05);
    ASSERT_NEAR(17.667, res, 0.05);
  }
}

TEST(multi_integration_monte_carlo, test4_multiplicity_is_3) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  std::vector<double>a = { 0.0, 2.5, 1.234 };
  std::vector<double>b = { 1.534, 3.12, 1.555 };
  double res = getIntegralMonteCarloParallel(f3, a, b, 1000000);
  if (rank == 0) {
    double res1 = getIntegralMonteCarloSequential(f3, a, b, 1000000);
    ASSERT_NEAR(res, res1, 0.05);
    ASSERT_NEAR(2.5, res, 0.05);
  }
}

TEST(multi_integration_monte_carlo, test5_multiplicity_is_4) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  std::vector<double>a = { 0.0, -2.5, 1.5, -5.0 };
  std::vector<double>b = { 1.0, -1.0, 2.5, -4.0 };
  double res = getIntegralMonteCarloParallel(f4, a, b, 1000000);
  if (rank == 0) {
```

```
    double res1 = getIntegralMonteCarloSequential(f4, a, b, 1000000);
    ASSERT_NEAR(res, res1, 0.05);
    ASSERT_NEAR(26.35, res, 0.05);
  }
}


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();
}
```