Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу "Численные методы"
по теме "Вычисление многократных интегралов с использованием квадратурных формул и метода Монте-Карло"

Студент: А. В. Куликов
Преподаватель: Ю. В. Сластушенский
Группа: М8О-308Б-17

Дата:
Оценка:
Подпись:

Москва 2020

1 Цель работы

Реализация, анализ, сравнительная характеристика, оценка погрешности метода многократного интегрирования с использованием квадратурных формул (метод трапеций) и метода Монте-Карло.

2 Оценка погрешности метода квадратурных формул

Необходимо вычислить интеграл

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_n \ldots dx_1$$

Введем вспомогательные функции $F_1, \ldots, F_n$, такие что

$$F_n(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \tag{1}$$

$$F_{i-1}(x_1, \ldots, x_{i-1}) = \int_{a_i}^{b_i} F_i(x_1, \ldots, x_i) dx_i \tag{2}$$

Тогда исходный интеграл будет равен

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_n \ldots dx_1 = \int_{a_1}^{b_1} F_1(x_1) dx_1$$

Каждый из интегралов можно заменить по квадратурной формуле метода трапеций суммой:

$$F_{i-1}(x_1, \ldots, x_{i-1}) = \int_{a_i}^{b_i} F_i(x_1, \ldots, x_i) dx_i = \sum_{k=1}^{k_i+1} F_i(x_1, \ldots, x_{i-1}^{(k)}) A_k h + R_i,$$

где R_i – остаточный член, после i-го интегрирования. Тогда, раскрывая с учетом (1), (2) получим

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_n \ldots dx_1 = \sum_{i_1=1}^{k_1+1} \cdots \sum_{i_n=1}^{k_n+1} C_{i_1,\ldots,i_n} f(x_1^{(i_1)}, \ldots, x_n^{(i_n)}) + R$$

где $C_{i_1,\ldots,i_n}$ – некий квадратурный коэффициент,

$$R = R_1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} k_j h^i R_{i+1}$$

– остаточный член приближения, $k_j = \left\lfloor \dfrac{b_j - a_j}{h} \right\rfloor$ – кол-во частей, на которое разделится отрезок интегрирования от a_j до b_j.

При этом для остаточного члена r при интегрирорвании методом трапеций функции $g(x)$ от c до d с шагом h известно:

$$r \leqslant \frac{\left|(d-c) \max\limits_{x \in [c,d]} \dfrac{d^2 g}{dx^2}(x)\right|}{12} h^2$$

По аналогии получаем:

$$R_i \leqslant \frac{\left|(b_i - a_i) \max\limits_{x_i \in [a_i, b_i]} \dfrac{\partial^2 F_i}{\partial x_i^2}(x_1, \ldots, x_i)\right|}{12} h^2 \tag{3}$$

при конкретных фиксированных $x_1, \ldots, x_{i-1}$.

С учетом (1), (2), и, используя теорему о дифференцировании по параметру под знаком интеграла, получим:

$$\frac{\partial^2 F_i}{\partial x_i^2}(x_1, \ldots, x_i) = \frac{\partial^2}{\partial x_i^2} \int\limits_{a_{i+1}}^{b_{i+1}} \cdots \int\limits_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_n \ldots dx_{i+1} =$$

$$= \int\limits_{a_{i+1}}^{b_{i+1}} \cdots \int\limits_{a_n}^{b_n} \frac{\partial^2 f}{\partial x_i^2}(x_1, \ldots, x_n) dx_n \ldots dx_{i+1}$$

при условии непрерываности функции f по всем параметрам и существовании частных производных $\dfrac{\partial f}{\partial x_1}, \ldots, \dfrac{\partial f}{\partial x_n}$ в области $V = [a_1, b_1] \times \cdots \times [a_n, b_n]$.

Введем число

$$M = \max_{i=1,\ldots,n} \left(\max_V \left| \frac{\partial^2 f}{\partial x_i^2} \right| \right)$$

тогда

$$\left| \frac{\partial^2 F_i}{\partial x_i^2} \right| \leqslant M \prod_{k=i+1}^{n} (b_k - a_k)$$

Т.о.

$$|R| = \left| R_1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} k_j \; h^i R_{i+1} \right| \leqslant$$

$$\leqslant \left| \frac{Mh^2}{12} \prod_{k=1}^{n} (b_k - a_k) + \sum_{i=1}^{n-1} \prod_{j=1}^{i} \left\lfloor \frac{b_j - a_j}{h} \right\rfloor \; h^i \prod_{k=i+1}^{n} (b_k - a_k) \frac{Mh^2}{12} \right| \leqslant$$

$$\leqslant \left| \frac{Mh^2}{12} \prod_{k=1}^{n} (b_k - a_k) + \sum_{i=1}^{n-1} \prod_{j=1}^{i} (b_j - a_j) \prod_{k=i+1}^{n} (b_k - a_k) \frac{Mh^2}{12} \right| =$$

$$= \left| \frac{Mh^2}{12} \sum_{i=1}^{n} \prod_{j=1}^{n} (b_j - a_j) \right| = \left| \frac{Mh^2 n}{12} \prod_{j=1}^{n} (b_j - a_j) \right| = \frac{Mh^2 n}{12} |V|$$

Итак, получаем

$$|R| \leqslant \frac{Mh^2 n}{12} |V| \tag{4}$$

– оценка сверху погрешности многократного интегрирования методом трапеций. Здесь h – минимальный шаг сетки необходимый для достижения точности $\varepsilon = |R|$. Многократное интегрирование методом трапеций, как и в одномерном случае, имеет точность порядка h^2.

Поиск точного значения числа M в общем случае задтруднителен и соизмерим с исходной задачей по вычислительной сложности. Поэтому, наверное, стоит подбирать M экспериментальным путем находя компромисс между временем вычислений и получаемой точностью.

3 Оценка погрешности метода Монте-Карло

Необходимо вычислить интеграл

$$I = \int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_n \ldots dx_1$$

Пусть ξ – случайный вектор, с плотностью вероятности

$$p(x) = \begin{cases} \dfrac{1}{\prod\limits_{i=1}^{n} (b_i - a_i)}, & \text{если } x \in [a_1, b_1] \times \cdots \times [a_n, b_n] \\ 0, & \text{в противном случае} \end{cases} \tag{5}$$

Введем случайную величину $\eta = \dfrac{f(\xi)}{p(\xi)}$, тогда

$$I = M_\eta = \int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} \frac{f(x_1, \ldots, x_n)}{p(x_1, \ldots, x_n)} p(x_1, \ldots, x_n) dx_n \ldots dx_1$$

Пусть $\eta^{(1)}, \ldots, \eta^{(k)}$ – реализации η. Они независимы и одинаково распределены. Тогда по ЦПТ случайная величина

$$X = \frac{\sum_{i=1}^{k} \eta^{(i)} - M_\eta k}{\sqrt{D_\eta k}} \sim N(0, 1)$$

Тогда

$$P(|X| \leqslant 3) = P\left(\left| \frac{\sum_{i=1}^{k} \eta^{(i)} - M_\eta k}{\sqrt{D_\eta k}} \right| \leqslant 3 \right) = P\left(\left| \frac{1}{k} \sum_{i=1}^{k} \eta^{(i)} - I \right| \leqslant 3\sqrt{\frac{D_\eta}{k}} \right)$$

С другой стороны

$$P(|X| \leqslant 3) = 2\Phi(3) = 0,9974$$

Т.о. с вероятностью $0,9974$ разница между выборочным средним $\overline{\eta} = \frac{1}{k} \sum_{i=1}^{k} \eta^{(i)}$ и настоящим значением интеграла I не превзойдет значения $3\sqrt{\frac{D_\eta}{k}}$.

Оценим теперь D_η. Пусть $\eta = \varphi(\xi) = \frac{f(\xi)}{p(\xi)}$. Разложим $\varphi(\xi)$ в ряд Тейлора до второго члена:

$$\varphi(\xi) = \varphi(\xi_1, \ldots, \xi_n) \approx \varphi(M_{\xi_1}, \ldots, M_{\xi_n}) + \sum_{i=1}^{n} \frac{\partial \varphi}{\partial \xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})(\xi_i - M_{\xi_i})$$

Тогда

$$D_\eta = D[\varphi(\xi)] \approx D[\varphi(M_{\xi_1}, \ldots, M_{\xi_n}) + \sum_{i=1}^{n} \frac{\partial \varphi}{\partial \xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})(\xi_i - M_{\xi_i})] =$$

$$= D[\varphi(M_{\xi_1}, \ldots, M_{\xi_n}) + \sum_{i=1}^{n} \frac{\partial \varphi}{\partial \xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})\xi_i - \sum_{i=1}^{n} \frac{\partial \varphi}{\partial \xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})M_{\xi_i}] =$$

$$= D[\sum_{i=1}^{n} \frac{\partial \varphi}{\partial \xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})\xi_i] = \sum_{i=1}^{n} \left(\frac{\partial \varphi}{\partial \xi_i}(M_{\xi_1}, \ldots, M_{\xi_n}) \right)^2 D_{\xi_i}$$

С учетом (5) получим

$$\varphi(\xi) = \frac{f(\xi)}{p(\xi)} = \prod_{i=1}^{n}(b_i - a_i)f(\xi)$$

$$\frac{\partial\varphi}{\partial\xi_i}(\xi_1, \ldots, \xi_n) = \prod_{i=1}^{n}(b_i - a_i)\frac{\partial f}{\partial\xi_i}(\xi_1, \ldots, \xi_n)$$

Также известно, что т.к. $D_{\xi_i} = \dfrac{(b_i - a_i)^2}{12}$. Тогда, получаем:

$$D_\eta \approx \frac{1}{12}\prod_{j=1}^{n}(b_j - a_j)^2 \sum_{i=1}^{n}\left[(b_i - a_i)\frac{\partial f}{\partial\xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})\right]^2$$

Имеем вероятностную оценку сверху погрешности вычислений

$$\delta \leqslant \frac{1}{2}\prod_{i=1}^{n}(b_i - a_i)\sqrt{\frac{3}{k}\sum_{i=1}^{n}\left[(b_i - a_i)\frac{\partial f}{\partial\xi_i}(M_{\xi_1}, \ldots, M_{\xi_n})\right]^2} \tag{6}$$

Из выражения видно, что точность вычислений обратно пропорциональна квадратному корню количеству пробных точек как и в одномерном случае.

4 Реализация

Получив выражения для оценки погрешности, и считая, что при заданных параметрах необходимая точность достигается (и она действительно достигается см. п. Сравнение и анализ) можно приступать к реализации. Прежде всего необходимо, зная точность, получить параметры для алгоритмов, при которых она будет достигнута.

4.1 Метод трапеций

Из (4) следует, что необходимый шаг сетки

$$h = \sqrt{\frac{12\varepsilon}{Mn\prod_{j=1}^{n}(b_j - a_j)}},$$

где ε – задаваемая точность.

Для практической реализации не возможно просто использовать полученное значение h как шаг сетки по всем отрезкам интегрирования из-за того, что, если отрезок не делится на целое количество подотрезков делением на h, то часть отрезка интегрирования останется неучтенной, и это приведет к значительной потере точности. Поэтому

для каждого отрезка интегрирования $[a_i, b_i]$ необходимо выбрать свой шаг $h_i \leqslant h$, для предотвращения потери точности.

Это можно сделать оптимально следующим образом:

$$h_i = \frac{b_i - a_i}{\left\lceil \dfrac{b_i - a_i}{h} \right\rceil}$$

Сам алгоритм был реализован рекурсивно для интеграла каждой вложенности, но это не должно приводить к большой потере производительности, т.к. никаких параметров рекурсивному вызову не передается, вся параметризация происходит через члены класса. На стеке при каждом вызове сохраняется только адрес следующей за вызовом команды в вызвавшей функции. С учетом экспоненциального роста вычислительной сложности от размерности задачи рекурсия не приведет к переполнению стека потому, т.к. максимальная размерность задачи, которую позволяет решать данный метод за приемлемое время сильно ограничена.

При каждом вызове просто подсчитывается и возвращается взвешенная сумма значений функций в наборе точек.

Не совсем ясно можно ли реализовать алгоритм итеративно т.к. это потребовало бы n жестко прописанных вложенных циклов, что исключает возможность параметризации алгоритма размерностью задачи n.

4.2 Метод Монте-Карло

Из (6) следует, что количество случайных точек для достижения заданной точности

$$k = \frac{9}{\varepsilon^2} D_\eta,$$

где ε – задаваемая точность.

Сам алгоритм предельно прост: подсчитываем сумму s значений функции в сгенерированых k точках, затем принимаем за результат значение $\dfrac{s}{k} \prod\limits_{i=1}^{n} (b_i - a_i)$.

Стоит упомянуть, что используется ГПСЧ `std::mt19937` вместо обычного `rand` т.к. генерируемые им числа “более случайны” и, к тому же, при включенных оптимизациях компилятора, он работает несколько быстрее.

Листинг кода на C++ приводится в приложении.

5 Пример работы

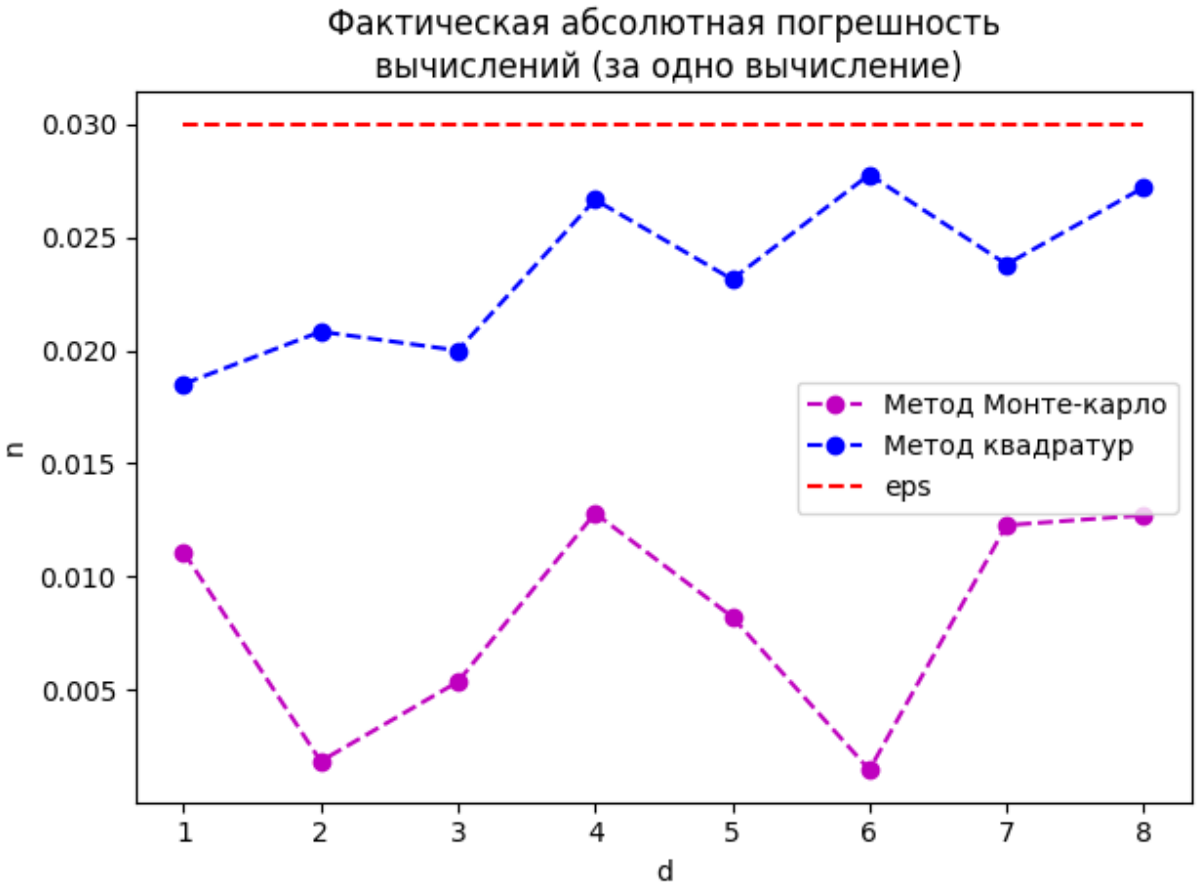
Пример запуска программы для вычисления интеграла

$$\int_0^3 \cdots \int_{2n}^{2n+3} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} x_i x_j + 1 \right) dx_n \dots dx_1$$

```
$ ./prog1
eps: 0.1
a: 0
b: 3
ref_res: 12
mc_res: 11.994771473994049
mc_err: 0.0052285260059505845
q_res: 12.091836734693876
q_err: 0.09183673469387621

a: 0 2
b: 3 5
ref_res: 247.5
mc_res: 247.529749953155
mc_err: 0.029749953155004505
q_res: 247.59342560553642
q_err: 0.093425605536424428

a: 0 2 4
b: 3 5 7
ref_res: 3064.5
mc_res: 3064.5196879226487
mc_err: 0.019687922648699896
q_res: 3064.5991836734706
q_err: 0.099183673470633948

a: 0 2 4 6
b: 3 5 7 9
ref_res: 26568
mc_res: 26567.922418177746
mc_err: 0.077581822253705468
q_res: 26568.099183673468
q_err: 0.099183673468360212
```

Здесь `eps` – задаваемая точность, `a` и `b` – границы интегрирования, `ref_res` – значение интеграла, вычисленное аналитически, `mc_res` и `q_res` – значения, полученные методом Монте-Карло и трапеций соответственно, `mc_res` и `q_res` – абсолютные погрешости вычислений, полученные методом Монте-Карло и трапеций соответственно.

Ниже приведен результат вычислений для другой функции.



Видно, что оценки погрешности состоятельны, и их можно использовать для вычисления оптимальных параметров для запуска алгоритма.

6 Сравнение и анализ

6.1 Сравнение погрешности

При одинаковой задаваемой точности вычислений метод Монте-Карло в среднем обеспечивает меньшую погрешность вычислений. Это обусловлено тем фактом, что погрешность имеет нормальное распределение с нулевым мат. ожиданием. Из-за этого гораздо вероятнее меньшая погрешность, чем большая, и тем более превышающая заданную точность. Погрешность же вычислений методом трапеций фиксирована.

Стоит отметить, что прмерно в $0,35\%$ запусков алгоритма Монте-Карло абсолютная погрешность по сравнению с аналитическим решением все же незначительно превышает задаваемую точность. Это можно было бы объяснить тем, что даваемая оценка является вероятностной, но тогда ожидаемый процент превышений был бы в районе $0,26\%$. Еще небольшой процент превышений добавляет использование не истинного

значения значения дисперсии D_η, а его приближенное значение. Это в случае функции, на которой проводились истпытания, приводит к небольшой недооценке дисперсии. Так истинное значение дисперсии есть $64,8$, а оценочное – $60,75$. Недооценка дисперсии приводит к недооценке числа k случайных точек, используемых для расчета, а это в свою очередь ведет к повышению процента испытаний, в которых наблюдается превышение допустимой погрешности.

6.2 Сравнение временых затрат

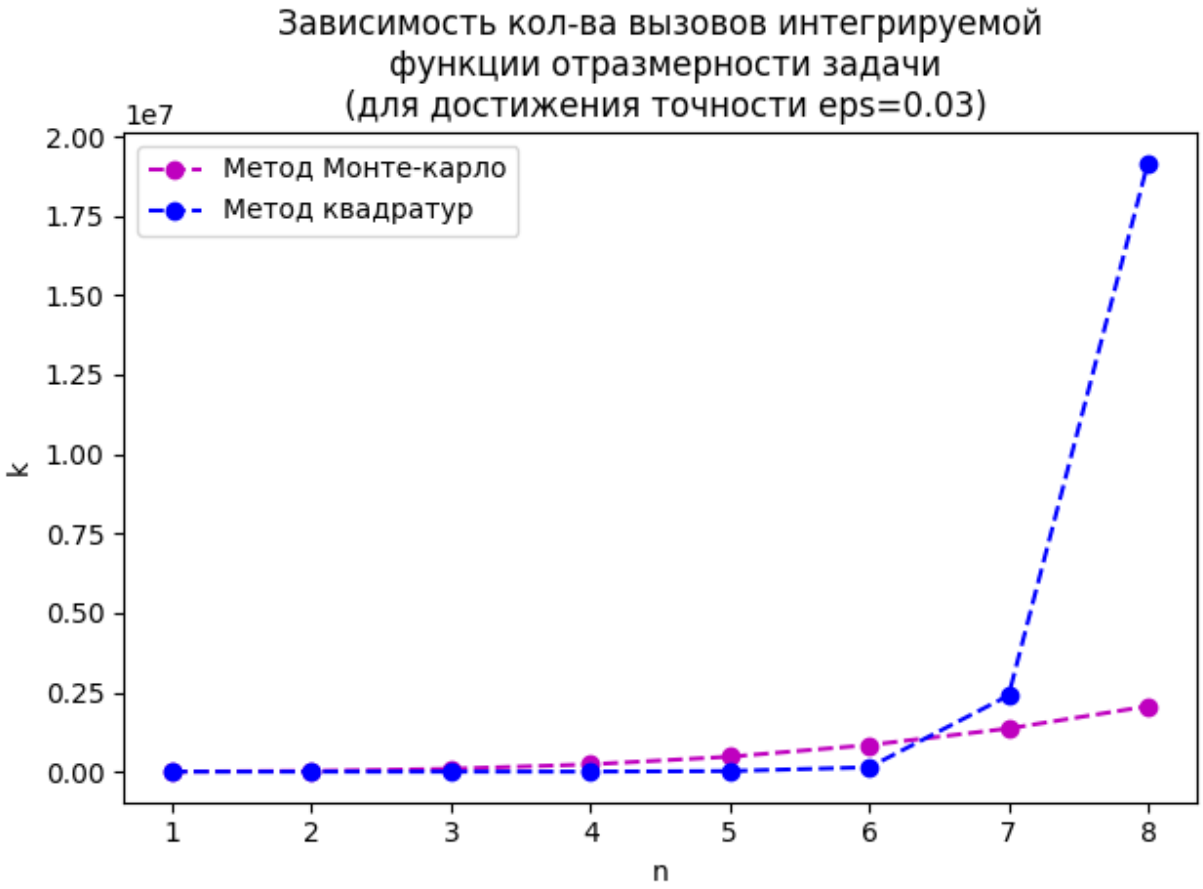
Вычислительным ядром обоих алгоритмов является вычисление значения функции в точке. Для обоих алгоритмов можно получить количесвто вычислений интегрируемой функции. Для метода Монте-Карло это число получено в разделе "Реализация". Если считать длину отрезка интегрирования $[a_i, b_i]$ одинаковой для всех интегралов и равной w, то число вычислений интегрируемой функциии для метода Монте-Карло можно считатать такой:

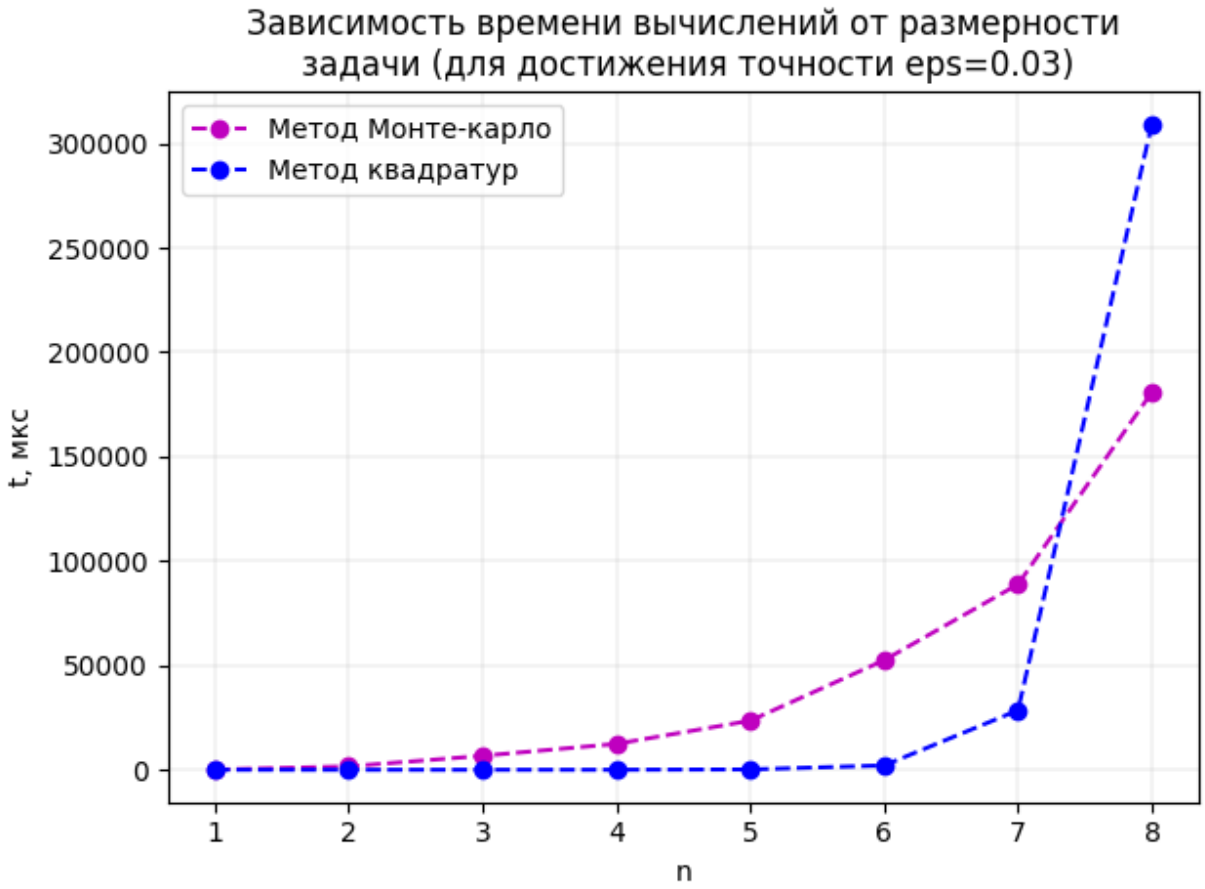
$$k_{MK} = \frac{9}{\varepsilon^2}\frac{1}{12}\prod_{i=1}^{n} w^2 \sum_{i=1}^{n} [cw]^2 = \frac{3nc^2}{4\varepsilon^2} w^{2n+2} = O(w^{2n+2})$$

Пусть теперь длина отрезка интегрирования w делится на вычисленный указанным способом шаг сетки h. Тогда функция будет вычислена в $\dfrac{w}{h} + 1$ точке на одно измерение. Всего измерений n, поэтому всего по всей области интегрирования функция будет вычислена в $\left(\dfrac{w}{h} + 1\right)^n$ точках. Тогда подставляя h имеем:

$$k_T = \left(\frac{w}{\sqrt{\dfrac{12\varepsilon}{Mnw^n}}} + 1 \right)^n \approx \left(w^{1+n/2} n^{1/2} \sqrt{\frac{M}{12\varepsilon}} \right)^n = O(d^{n+n^2/2}\ n^{n/2})$$

Если принять, что каждое вычисление функции происходит за $O(1)$, то такой же порядок возрастание будет иметь и время, требуемое на вычисление интеграла. Очевидно, что при некотором n количество вызовов функции в методе трапеций начинает превосходить количество вызовов в методе Монте-Карло. Это же подтверждается и экспериментом.

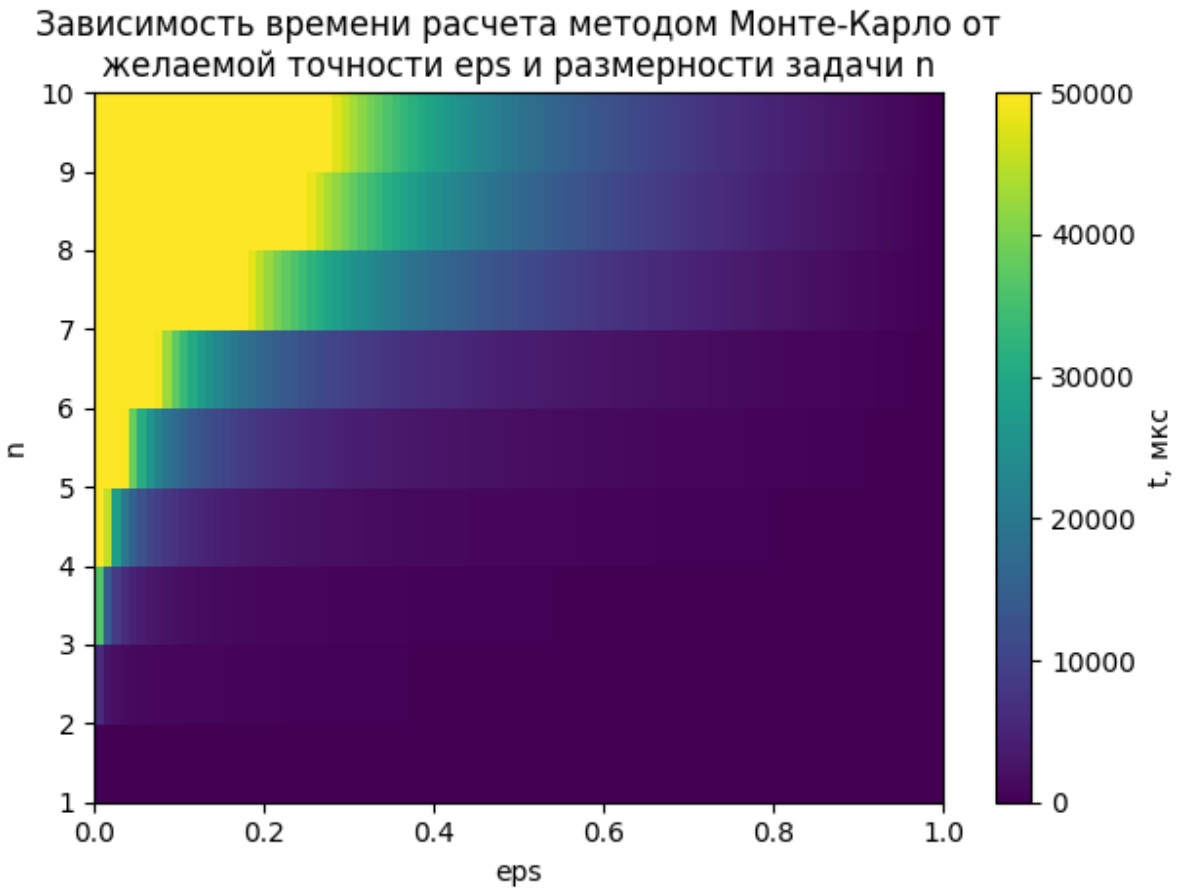

Зависимость кол-ва вызовов интегрируемой
функции отразмерности задачи
(для достижения точности eps=0.03)
1e7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
k
Метод Монте-карло
Метод квадратур
1
2
3
4
5
6
7
8
n

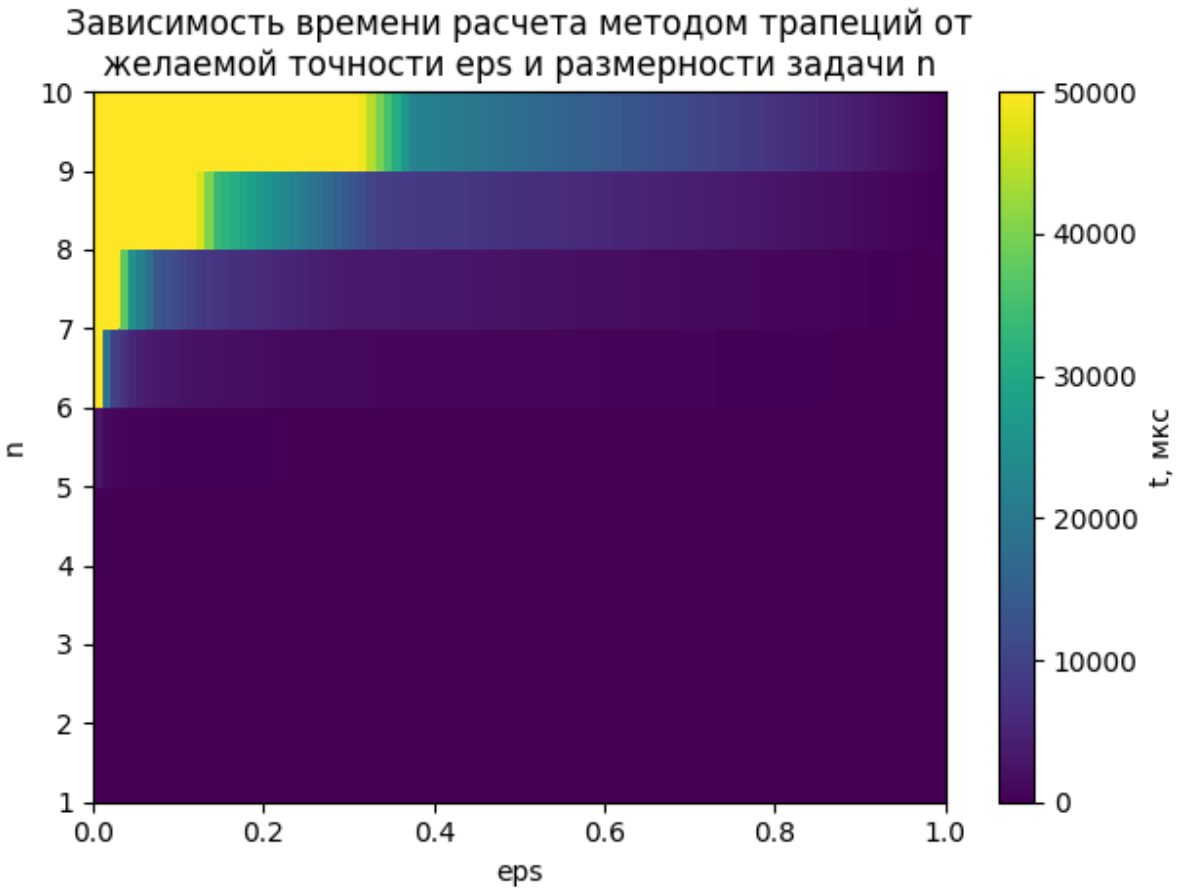


Точка пересечения на графике с временем слегка “запаздывает”. Это объясняется тем, что в методе Монте-Карло при повышении размерности увеличивается не только количество случайных точек, но и количество координат в каждой из точек. Поэтому количество генерируемых случайных чисел растет даже быстрее, чем количество точек. И затрачиваемое на это время нельзя не учитывать. Профайлер показывает, что, хотя на вычисление функции тратится больше времени работы программы (51%), чем на генерацию точек (36%), но все равно это соизмеримые величины.

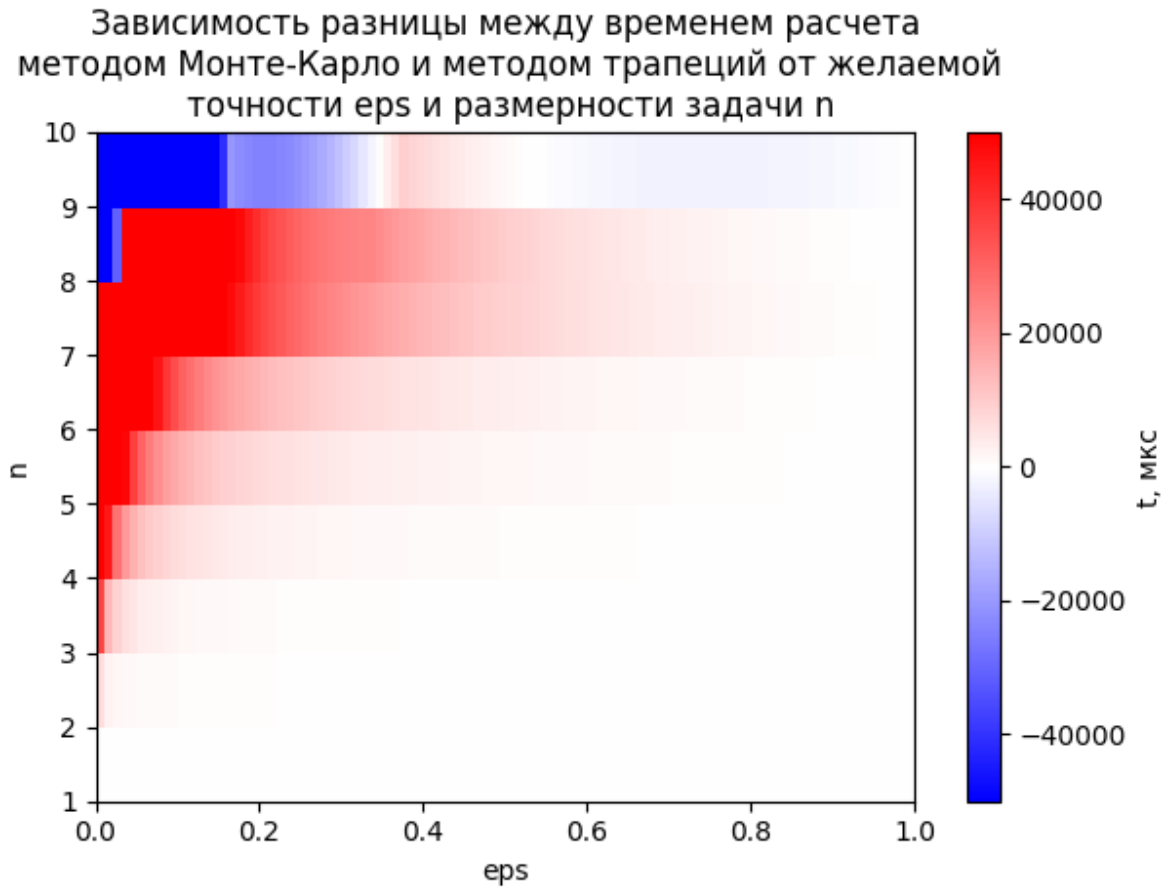
Как и ожидалось, в задачах меньшей размерности выгоднее использовать метод трапеций, но с ростом размерности стоит отдавать предпочтение методу Монте-Карло.

Затрачиваемое время при различных точностях и размерностях приводится на следующих графиках:




Зависимость времени расчета методом трапеций от
желаемой точности eps и размерности задачи n
n
eps
t, мкс
10
9
8
7
6
5
4
3
2
1
0.0
0.2
0.4
0.6
0.8
1.0
50000
40000
30000
20000
10000
0

Так же на графике ниже достаточно наглядно показаны параметры, при которых один метод превосходит другой.



В синей области выгоднее использовать метод Монте-Карло, в красной – метод трапеций, в белой области, соответственно, не принципиально в данном масштабе.

7 Заключение

Многократное интегрирование – это одна из задач, в которой проявляется так называемое «Проклятие размерности». С ростом кратности интеграла значительно возрастает время, необходимое для достижения заданной точности решения задачи. Поэтому на передний план выходит выбор алгоритма многократного интегрирования. Было выяснено, что при малых размерностях метод трапеций значительно опережает метод Монте-Карло, но при больших размерностях картина кардинально меняется: уже при кратности интеграла 7-8 алгоритм Монте-Карло начинает существенно превосходить метод трапеций по скорости вычисления.

В ходе работы над курсовым проектом были получены оценки погрешности двух численных методов многомерного интегрирования: метод трапеций и метод Монте-

Карло. Каждый из методов был реализован, проведен анализ каждого каждого из методов и приведена их сравниетльная характеристика.

Список источников

1. «Методические указания к решению задач по численному интегрированию» Калашников А.Л., Федоткин А.М., Фокина В.Н. 2016.

2. Статья «Метод Монте-Карло и его точность»: https://habr.com/ru/post/274975/

3. Статья «Approximating the expected value and variance of the function of a (continuous univariate) random variable»: https://stats.stackexchange.com/questions/301861/approximating-the-expected-value-and-variance-of-the-function-of-a-continuous-u

Приложение

Листинг кода

multidim_integral.h

```
#ifndef INTEGRAL_H
#define INTEGRAL_H

#include <vector>
#include <cmath>
#include <cassert>
#include <random>
#include <cstdlib>

#include "deriv.h"

double random_double(double a, double b){
    static std::mt19937 gen(time(0));
    static const double norm_coef = 1.0 / gen.max();

    return a + (double)gen() * (b - a) * norm_coef;
}

void random_vector(const std::vector<double> &a, const std::vector<
    double> &b, std::vector<double> &v){
    size_t n = v.size();
    for(size_t i = 0; i < n; ++i){
        v[i] = random_double(a[i], b[i]);
    }
}

template<class F>
double monte_carlo_method(const F &f, const std::vector<double> &a,
    const std::vector<double> &b, unsigned long long k){
    size_t n = a.size();
    assert(n == b.size());
    std::vector<double> x(n);
    double s = 0.0;

    double v = 1.0;
    for(size_t i = 0; i < n; ++i){
        v *= b[i] - a[i];
    }

    for(unsigned long long i = 0; i < k; ++i){
        random_vector(a, b, x);
        s += f(x);
    }

    return s * v / k;
}
```

```
45
46 template<class F>
47 double monte_carlo_prec(const F &f, const std::vector<double> &a,
       const std::vector<double> &b, double eps){
48     size_t n = a.size();
49     double p = 1.0;
50     double s = 0.0;
51
52     std::vector<double> x(n);
53     for(size_t i = 0; i < n; ++i)
54         x[i] = (a[i] + b[i]) / 2.0;
55
56     for(size_t i = 0; i < n; ++i){
57         double t = b[i] - a[i];
58         p *= t * t;
59         double d = simple_partial_deriv(f, x, 0.001, i);
60
61         s += d*d * t*t;
62     }
63
64     double D = p * s / 12.0;
65
66     unsigned long long k = ceil(9.0 * D/(eps * eps));
67
68     return monte_carlo_method(f, a, b, k);
69 }
70
71 template<class F>
72 class Quadrature{
73 private:
74     const F &_f;
75     std::vector<double> _a, _b, _h;
76     double sum;
77     double _M;
78     double rh;
79
80     std::vector<double> x;
81     unsigned int k;
82 double aux(){
83         double s = 0.0;
84         double w = _b[k] - _a[k];
85         unsigned int c = round(w / _h[k]);
86         if(k < _a.size()-1){
87             ++k;
88             s += 0.5 * aux();
89             for(unsigned int i = 1; i < c; ++i){
90                 x[k] = _a[k] + i * _h[k];
91                 ++k;
92                 s += aux();
93             }
94             x[k] = _b[k];
```

```
95              ++k;
96              s += 0.5 * aux();
97          }
98          else{
99              s += 0.5 * _f(x);
100             for(unsigned int i = 1; i < c; ++i){
101                 x[k] = _a[k] + i * _h[k];
102                 s += _f(x);
103             }
104             x[k] = _b[k];
105             s += 0.5 * _f(x);
106         }
107         s *= _h[k];
108
109         x[k] = _a[k];
110         --k;
111
112         return s;
113     }
114
115     double proper_h(double eps, double M){
116         size_t n = _a.size();
117         double V = 1.0;
118         for(unsigned int i = 0; i < n; ++i){
119             V *= _b[i] - _a[i];
120         }
121
122         return sqrt(12.0 * eps / (M * n * V));
123     }
124
125 public:
126     Quadrature(const F &f, std::vector<double> &a, std::vector<double>
            &b, double eps, double M)
127     : _f(f), _a(a), _b(b), _M(M) {
128         size_t n = a.size();
129         rh = proper_h(eps, _M);
130         _h = std::vector<double>(n);
131
132
133         for(size_t i = 0; i < n; ++i){
134             double w = _b[i] - _a[i];
135             _h[i] = w / ceil(w / rh);
136         }
137
138         x = _a;
139         k = 0;
140         sum = aux();
141     }
142
143     double integral(){
144         return sum;
```

```
    }

    double err(){
        size_t n = _a.size();
        double p = 1.0;
        for(size_t i = 0; i < n; ++i){
            p *= _b[i] - _a[i];
        }

        return _M * rh * rh * n * p / 12.0;
    }
};

#endif
```

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <vector>

#include <cmath>
#include "../include/multdim_integral.h"

using namespace std;

double f(const std::vector<double> &v){
    double sum = 0.0;
    for(auto i : v){
        for(auto j : v){
            sum += i * j;
        }
    }
    return sum + 1.0;
}

int main(){
    srand(time(0));
    rand();

    double eps = 0.1;
    double M = 2.0;
    vector<double> ref_values{12.0, 495.0/2.0, 6129.0/2.0, 26568.0};

    cout << "eps: " << eps << endl;

    for(unsigned int k = 1; k <= 4; ++k){
        vector<double> a(k), b(k);
        for(unsigned int i = 0; i < k; ++i){
            a[i] = 2 * i;
            b[i] = 2 * i + 3;
        }
```

```
36
37          cout << "a: ";
38          for(auto i : a)
39              cout << i << ' ';
40          cout << endl;
41
42          cout << "b: ";
43          for(auto i : b)
44              cout << i << ' ';
45          cout << endl;
46
47          cout.precision(17);
48
49          double mc_res = monte_carlo_prec(f, a, b, eps);
50          double err = abs(mc_res - ref_values[k-1]);
51
52          cout << "ref_res: " << ref_values[k-1] << endl;
53
54          cout << "mc_res: " << mc_res << endl;
55          cout << "mc_err: " << err << endl;
56
57
58          Quadrature<decltype(f)> q(f, a, b, eps, M);
59          double q_res = q.integral();
60          cout << "q_res: " << q_res << endl;
61          cout << "q_err: " << abs(q_res - ref_values[k-1]) << endl;
62          cout << endl;
63      }
64  }
```