Отчет по задаче A1

Гуршумов Даниил Бахтиятович
БПИ 239

Аннотация

брбрбр 3 бзбзбз 1
В данном отчете представлен алгоритм Монте-Карло для приближенного вычисления площади пересечения трех кругов. Проведены экспериментальные замеры точности оценки площади в зависимости от масштаба прямоугольной области и количества случайных точек. Представлены графики, анализ и выводы по полученным результатам.

1 Постановка задачи

- Реализовать алгоритм Монте-Карло для оценки площади пересечения трех кругов, заданных координатами центров и радиусами.
- Исследовать точность метода в зависимости от:
  1. масштаба прямоугольной области;
  2. количества случайно сгенерированных точек N, варьирующегося от 100 до 100,000 с шагом 500.
- Построить два типа графиков:
  - приближенное значение площади от параметров;
  - относительное отклонение от точного значения площади.
- Сделать выводы.

2 Реализация алгоритма

2.1 Код на C++

Для расчета площади использован следующий код:

```cpp
#include <iostream>
#include <cmath>
#include <random>
#include <vector>
#include <fstream>
#include <iomanip>

std::vector<double> Intersection(std::vector<double> X, std::vector<double> Y,
    std::vector<double> Z, int points) {
    double min_x = std::min({X[0] - X[2], Y[0] - Y[2], Z[0] - Z[2]});
    double max_x = std::max({X[0] + X[2], Y[0] + Y[2], Z[0] + Z[2]});
    double min_y = std::min({X[1] - X[2], Y[1] - Y[2], Z[1] - Z[2]});
    double max_y = std::max({X[1] + X[2], Y[1] + Y[2], Z[1] + Z[2]});
    double area_wide = (max_x - min_x) * (max_y - min_y);

    double tight_min_x = std::max({X[0] - X[2], Y[0] - Y[2], Z[0] - Z[2]});
    double tight_max_x = std::min({X[0] + X[2], Y[0] + Y[2], Z[0] + Z[2]});
    double tight_min_y = std::max({X[1] - X[2], Y[1] - Y[2], Z[1] - Z[2]});
    double tight_max_y = std::min({X[1] + X[2], Y[1] + Y[2], Z[1] + Z[2]});
    double area_tight = (tight_max_x - tight_min_x) * (tight_max_y -
        tight_min_y);

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> distr_x(min_x, max_x);
    std::uniform_real_distribution<> distr_y(min_y, max_y);
    std::uniform_real_distribution<> distr_x_t(tight_min_x, tight_max_x);
    std::uniform_real_distribution<> distr_y_t(tight_min_y, tight_max_y);

    int count_wide = 0;
    int count_tight = 0;

    for (int i = 0; i < points; ++i) {
        double rand_x = distr_x(gen);
        double rand_y = distr_y(gen);
        double rand_x_t = distr_x_t(gen);
        double rand_y_t = distr_y_t(gen);

        bool is_inside1 = (rand_x - X[0]) * (rand_x - X[0]) + (rand_y - X[1]) *
             (rand_y - X[1]) <= X[2] * X[2];
        bool is_inside2 = (rand_x - Y[0]) * (rand_x - Y[0]) + (rand_y - Y[1]) *
             (rand_y - Y[1]) <= Y[2] * Y[2];
        bool is_inside3 = (rand_x - Z[0]) * (rand_x - Z[0]) + (rand_y - Z[1]) *
             (rand_y - Z[1]) <= Z[2] * Z[2];

        bool is_inside1_t = (rand_x_t - X[0]) * (rand_x_t - X[0]) + (rand_y_t -
             X[1]) * (rand_y_t - X[1]) <= X[2] * X[2];
        bool is_inside2_t = (rand_x_t - Y[0]) * (rand_x_t - Y[0]) + (rand_y_t -
             Y[1]) * (rand_y_t - Y[1]) <= Y[2] * Y[2];
        bool is_inside3_t = (rand_x_t - Z[0]) * (rand_x_t - Z[0]) + (rand_y_t -
             Z[1]) * (rand_y_t - Z[1]) <= Z[2] * Z[2];

        if (is_inside1 && is_inside2 && is_inside3) {
            count_wide++;
        }

        if (is_inside1_t && is_inside2_t && is_inside3_t) {
            count_tight++;
        }
    }

    double intersection_area_wide = (static_cast<double>(count_wide) / points)
        * area_wide;
    double intersection_area_tight = (static_cast<double>(count_tight) / points
        ) * area_tight;
    return {intersection_area_wide, intersection_area_tight};
}

int main() {
    std::vector<double> X {1.0, 1.0, 1.0};
    std::vector<double> Y {1.5, 2.0, (std::sqrt(5) / 2.0)};
    std::vector<double> Z {2.0, 1.5, (std::sqrt(5) / 2.0)};

    int minPoints = 100;
    int maxPoints = 100000;
    int step = 500;

    std::ofstream outFile("results.csv");
    outFile << "Points,Area_wide,Area_tight\n";

    for (int points = minPoints; points <= maxPoints; points += step) {
        std::vector<double> area = Intersection(X, Y, Z, points);

        outFile << points << "," << area[0] << "," << area[1] << "\n";

        std::cout << "Points: " << points
                  << " | Wide: " << std::setprecision(8) << area[0]
                  << " | Tight: " << area[1] << "\n";
    }

    outFile.close();
    std::cout << "                    |                  |  |        results.
        csv" << std::endl;

    return 0;
}
```

2.2 Вывод данных

Код сохраняет результаты расчета в CSV-файл в формате:

$$\text{Points}, \text{Area\_wide}, \text{Area\_tight}$$

3 Результаты экспериментов

3.1 График 1: Приближенное значение площади

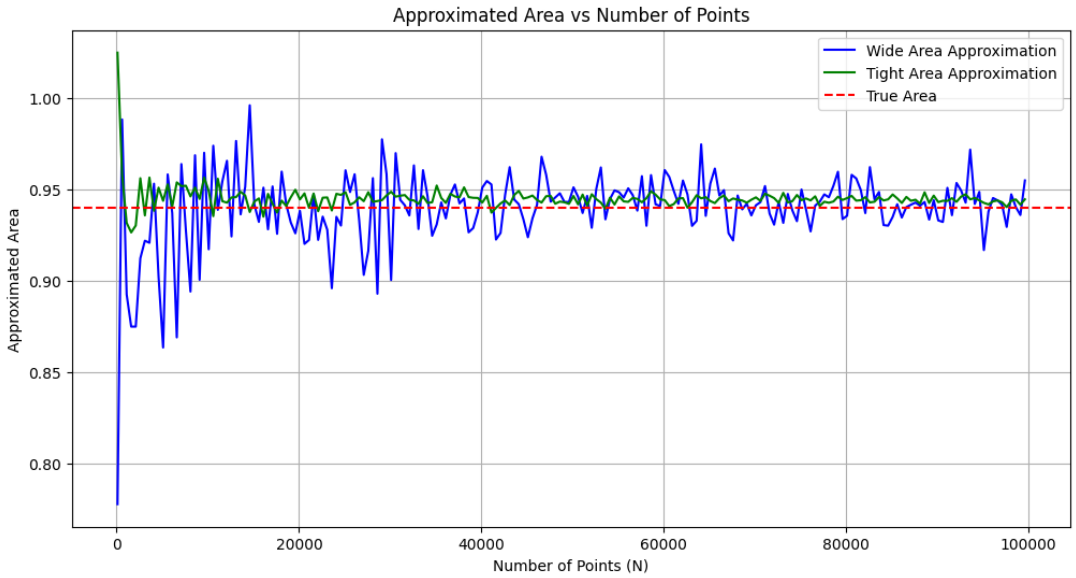

Рис. 1: Зависимость приближенной площади от количества точек N.

3.2 График 2: Относительное отклонение

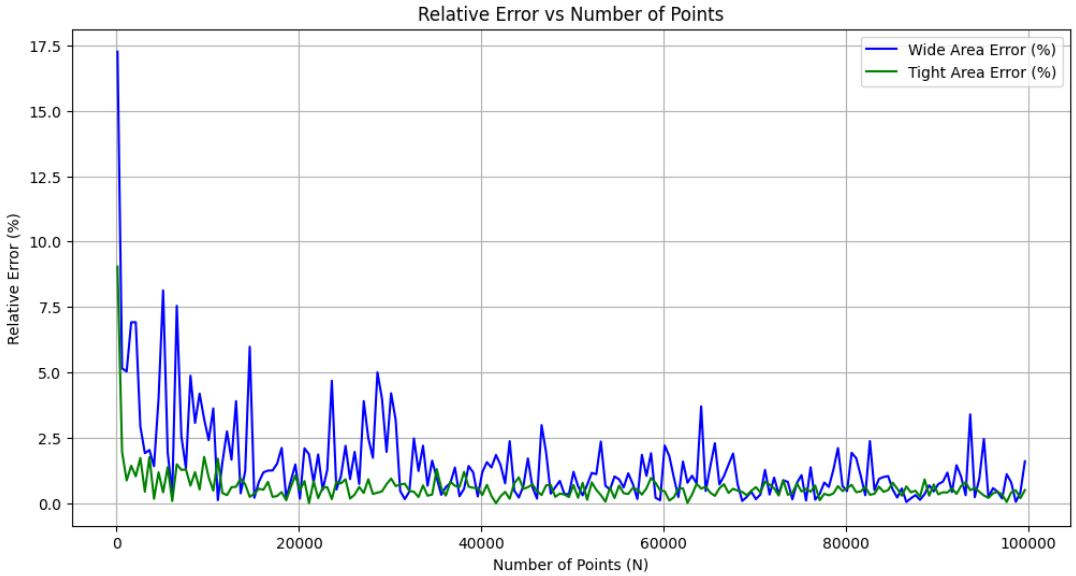

Рис. 2: Зависимость относительного отклонения от точного значения площади.

4 Анализ и выводы

1. **Приближенная площадь:**
   - С увеличением количества точек N приближенная площадь стабилизируется вокруг истинного значения.
   - Использование "тесной"области (tight area) приводит к более точным результатам на малых значениях N.
2. **Относительное отклонение:**
   - Относительное отклонение уменьшается с увеличением N, что подтверждает сходимость метода Монте-Карло.
   - На графике видно, что отклонение для "широкой"области (wide area) выше из-за большего объема случайных точек вне пересечения кругов.
3. **Общий вывод:**
   - Метод Монте-Карло является эффективным для оценки площади пересечения кругов.
   - Для повышения точности при фиксированном количестве точек рекомендуется уменьшать масштаб области генерации.