Отчет a1

ID посылки: 349088429
GitHub: https://github.com/sshpv/Algorithms_SE2/tree/main/A1

- `a1.cpp` - общее решение, выводит строки формата `N, S_wide, S_narrow` - где `S_wide` / `S_narrow` — оценки площади пересечения трёх кругов, полученные при генерации точек в широкой / узкой области соответственно.
- `a1i.cpp` - алгоритм из CodeForces
- `report.pdf` - отчет
- `data.txt` - строки формата `N, S_wide, S_narrow`, т. е. результаты полученные при соответствующем `N` с помощью 1го и 2г способов (с этим файлом я как раз работаю в google colab для составления графиков)
- `graphs.ipynb` - колабовский файл с построением графиков

Код с уже имеющимися данными (a1)

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

struct Circle {
    double cx;
    double cy;
    double r;
};

struct Box {
    double xmin;
    double xmax;
    double ymin;
    double ymax;
};

bool insideCircle(const Circle& c, double x, double y) {
    const double dx = x - c.cx;
    const double dy = y - c.cy;
    return dx * dx + dy * dy <= c.r * c.r;
}

bool insideAll(const std::vector<Circle>& circles, double x, double y) {
    for (const auto& c : circles) {
```

```
        if (!insideCircle(c, x, y)) {
            return false;
        }
    }
    return true;
}

double estimateArea(const Box& box,
                    const std::vector<Circle>& circles,
                    int samples,
                    std::mt19937_64& rng)
{
    std::uniform_real_distribution<double> distX(box.xmin, box.xmax);
    std::uniform_real_distribution<double> distY(box.ymin, box.ymax);

    int hits = 0;
    for (int i = 0; i < samples; ++i) {
        double x = distX(rng);
        double y = distY(rng);
        if (insideAll(circles, x, y)) {
            ++hits;
        }
    }

    const double boxArea = (box.xmax - box.xmin) * (box.ymax - box.ymin);
    return boxArea * static_cast<double>(hits) / static_cast<double>
(samples);
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    const double root5 = std::sqrt(5.0);

    Circle c1{1.0, 1.0, 1.0};
    Circle c2{1.5, 2.0, root5 / 2.0};
    Circle c3{2.0, 1.5, root5 / 2.0};

    std::vector<Circle> circles = {c1, c2, c3};

    Box wide{};
    wide.xmin = std::min({c1.cx - c1.r, c2.cx - c2.r, c3.cx - c3.r});
    wide.xmax = std::max({c1.cx + c1.r, c2.cx + c2.r, c3.cx + c3.r});
    wide.ymin = std::min({c1.cy - c1.r, c2.cy - c2.r, c3.cy - c3.r});
    wide.ymax = std::max({c1.cy + c1.r, c2.cy + c2.r, c3.cy + c3.r});

    Box narrow{};
    narrow.xmin = std::max({c1.cx - c1.r, c2.cx - c2.r, c3.cx - c3.r});
    narrow.xmax = std::min({c1.cx + c1.r, c2.cx + c2.r, c3.cx + c3.r});
```

```
    narrow.ymin = std::max({c1.cy - c1.r, c2.cy - c2.r, c3.cy - c3.r});
    narrow.ymax = std::min({c1.cy + c1.r, c2.cy + c2.r, c3.cy + c3.r});

    if (narrow.xmin >= narrow.xmax || narrow.ymin >= narrow.ymax) {
        std::cerr << "Error: narrow box is empty.\n";
        return 1;
    }

    std::mt19937_64 rng(123456789);

    std::cout << std::fixed << std::setprecision(8);

    for (int N = 100; N <= 100000; N += 500) {
        double S_wide   = estimateArea(wide,   circles, N, rng);
        double S_narrow = estimateArea(narrow, circles, N, rng);

        std::cout << N << ' '
                  << S_wide << ' '
                  << S_narrow << '\n';
    }

    return 0;
}
```

Изначальный код для задачи a1i - универсальная версия алгоритма Монте-Карло

```
#include <iostream>
#include <random>
#include <cmath>
#include <algorithm>
#include <iomanip>

using namespace std;

struct Circle {
    double x;
    double y;
    double r2;
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    Circle c1, c2, c3;
    double r;
```

```
    if (!(cin >> c1.x >> c1.y >> r)) return 0;
    c1.r2 = r * r;
    if (!(cin >> c2.x >> c2.y >> r)) return 0;
    c2.r2 = r * r;
    if (!(cin >> c3.x >> c3.y >> r)) return 0;
    c3.r2 = r * r;

    double xmin = max({c1.x - sqrt(c1.r2), c2.x - sqrt(c2.r2), c3.x -
sqrt(c3.r2)});
    double xmax = min({c1.x + sqrt(c1.r2), c2.x + sqrt(c2.r2), c3.x +
sqrt(c3.r2)});
    double ymin = max({c1.y - sqrt(c1.r2), c2.y - sqrt(c2.r2), c3.y -
sqrt(c3.r2)});
    double ymax = min({c1.y + sqrt(c1.r2), c2.y + sqrt(c2.r2), c3.y +
sqrt(c3.r2)});

    if (xmin >= xmax || ymin >= ymax) {
        cout << fixed << setprecision(20) << 0.0 << '\n';
        return 0;
    }

    double width  = xmax - xmin;
    double height = ymax - ymin;
    double rectArea = width * height;

    const int SAMPLES = 2000000;

    mt19937_64 rng(123456789);
    uniform_real_distribution<double> dist01(0.0, 1.0);

    auto inside_all = [&](double x, double y) -> bool {
        double dx1 = x - c1.x, dy1 = y - c1.y;
        if (dx1 * dx1 + dy1 * dy1 > c1.r2) return false;

        double dx2 = x - c2.x, dy2 = y - c2.y;
        if (dx2 * dx2 + dy2 * dy2 > c2.r2) return false;

        double dx3 = x - c3.x, dy3 = y - c3.y;
        if (dx3 * dx3 + dy3 * dy3 > c3.r2) return false;

        return true;
    };

    long long hits = 0;

    for (int i = 0; i < SAMPLES; ++i) {
        double rx = dist01(rng);
        double ry = dist01(rng);
        double x = xmin + rx * width;
        double y = ymin + ry * height;
```

```
            if (inside_all(x, y)) {
                ++hits;
            }
        }

        double p = static_cast<double>(hits) / static_cast<double>(SAMPLES);
        double area_est = rectArea * p;

        cout << fixed << setprecision(20) << area_est << '\n';
        return 0;
    }
```

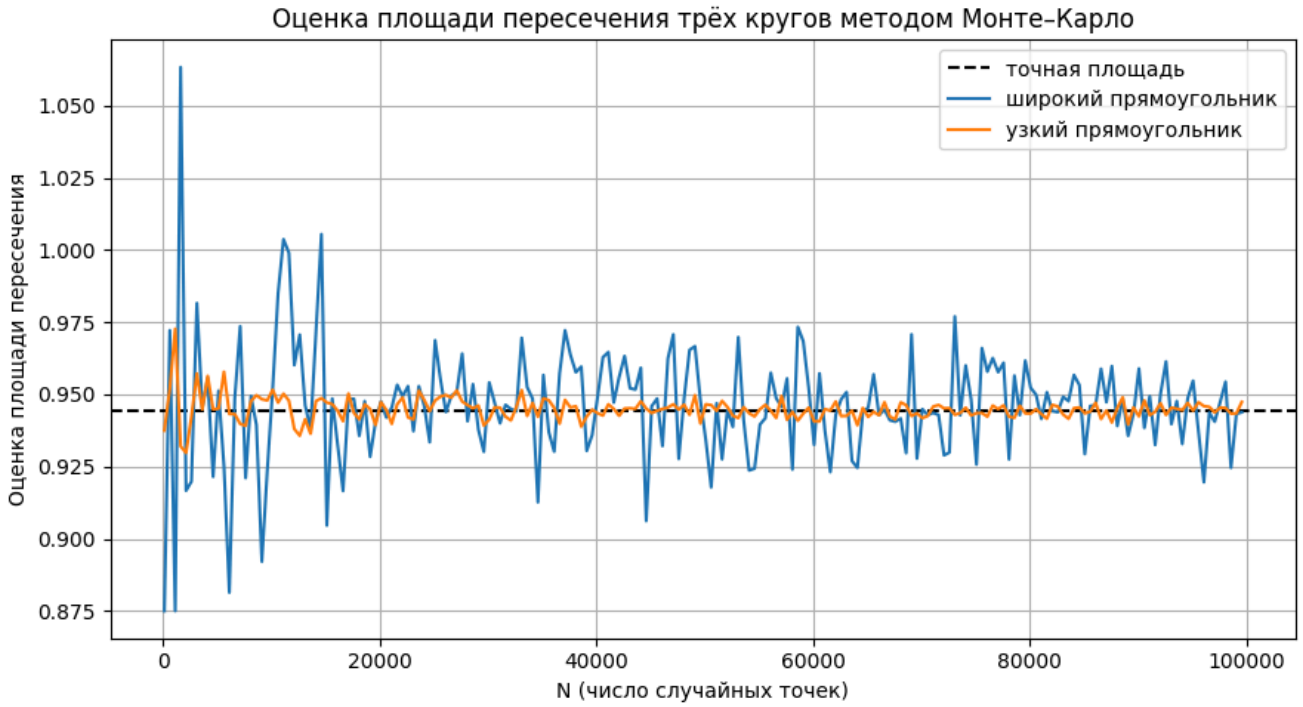

График 1: Зависимость оценённой площади пересечения трёх кругов от числа сгенерированных точек N для широкой и узкой прямоугольных областей. Видно, что при росте N обе оценки сходятся к точному значению площади, при этом для узкой области разброс значений меньше, чем для широкой.

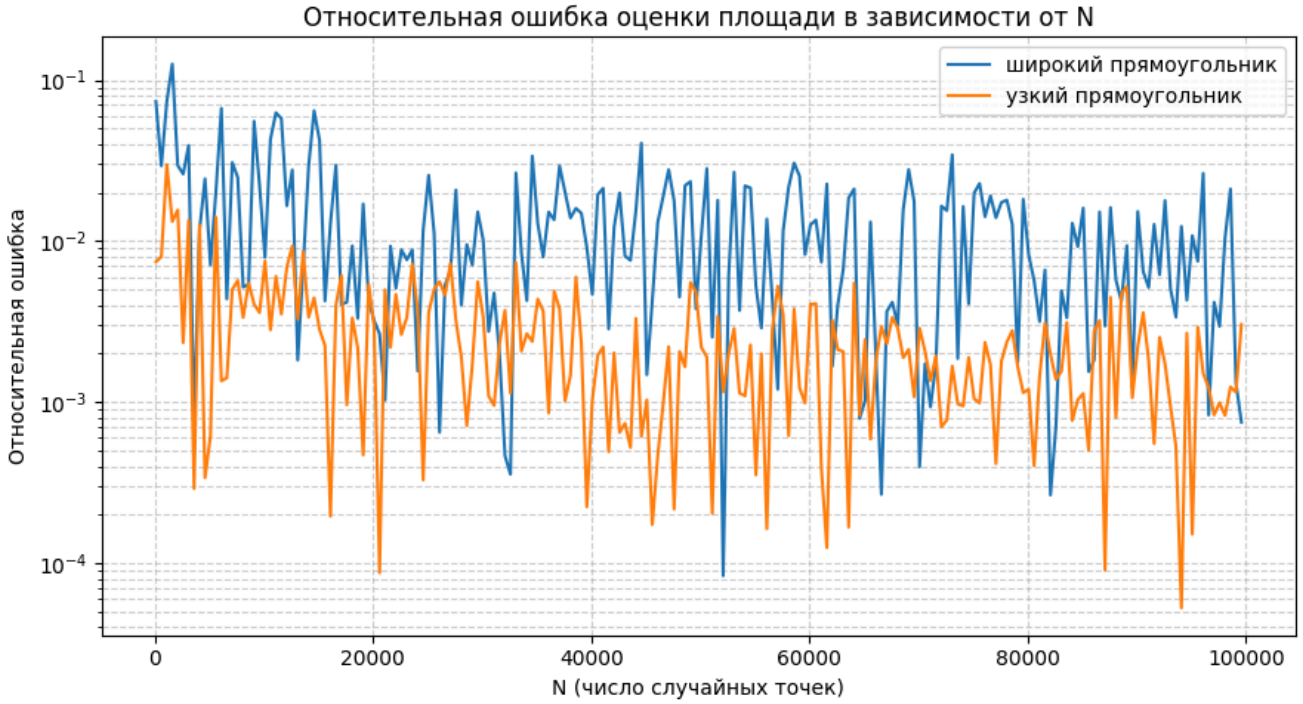

График 2: Зависимость относительной ошибки оценки площади от числа точек N (по оси ординат использован логарифмический масштаб). Для обеих областей ошибка в среднем убывает примерно как $1/\sqrt{N}$, причём для узкой области она на всём диапазоне N заметно ниже, чем для широкой.

Выводы

- Метод Монте-Карло корректно восстанавливает площадь пересечения трёх заданных кругов, приближаясь к аналитическому значению
- Качество оценки определяется числом точек N и тем, насколько «плотно» выбран прямоугольник, в котором моделируются точки. Чем ближе площадь прямоугольника к площади самой фигуры, тем эффективнее метод - такой вывод можно сделав оценив графики выше