Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра прикладной математики

Лабораторная работа №1
по дисциплине «Уравнения математической физики»

Решение эллиптических краевых задач методом конечных разностей

Факультет: ПМИ
Группа: ПМ-63
Студент: Шепрут И.И.
Вариант: 10
Преподаватель: Патрушев И.И.

Новосибирск
2019

1 Цель работы

Разработать программу решения эллиптической краевой задачи методом конечных разностей. Протестировать программу и численно оценить порядок аппроксимации.

2 Задание

1. Построить прямоугольную сетку в области в соответствии с заданием. Допускается использовать фиктивные узлы для сохранения регулярной структуры.
2. Выполнить конечноразностную аппроксимацию исходного уравнения в соответствии с заданием. Получить формулы для вычисления компонент матрицы A и вектора правой части b.
3. Реализовать программу решения двумерной эллиптической задачи методом конечных разностей с учетом следующих требований:
   - язык программирования C++ или Фортран;
   - предусмотреть возможность задания неравномерных сеток по пространству, учет краевых условий в соответствии с заданием;
   - матрицу хранить в диагональном формате, для решения СЛАУ использовать метод блочной релаксации [2, стр. 886], [3].
4. Протестировать разработанные программы на полиномах соответствующей степени.
5. Провести исследования порядка аппроксимации реализованного методов для различных задач с неполиномиальными решениями. Сравнить полученный порядок аппроксимации с теоретическим.

Вариант 10: Область имеет Ш-образную форму. Предусмотреть учет первых и третьих краевых условий.

3 Анализ задачи

Необходимо решить задачу:

$$\Delta u = f(x, y)$$

Первые краевые условия записываются в виде: $u(x, y)|_\Gamma = g_1(x, y)$, где $g_1(x, y)$ — известная функция.

Третьи краевые условия записываются в виде: $u'(x, y) + A \cdot u(x, y)|_\Gamma = g_3(x, y)$ где $g_3(x, y)$ — известная функция.

На первом этапе решения задачи нужно построить сетку. Матрица формируется одним проходом по всем узлам, для регулярных узлов заполняется согласно пятиточечному шаблону, для прочих – в соответствии с краевыми условиями.

4 Таблицы и графики

Параметры:

- Сетка имеет Ш-образную форму.
- Область сетки: $[0, 1] \times [0, 1]$.
- Количество элементов: 20 по обеим осям.
- Сетка равномерная.

● Точностью решения является норма L_2 разности векторов истинных значений и значений, вычисленных с помощью конечно-разностной аппроксимации.
● СЛАУ решается при помощи метода Гаусса-Зейделя с точностью 10^{-12}, максимальным числом итераций 3000 и параметром релаксации 1.4.

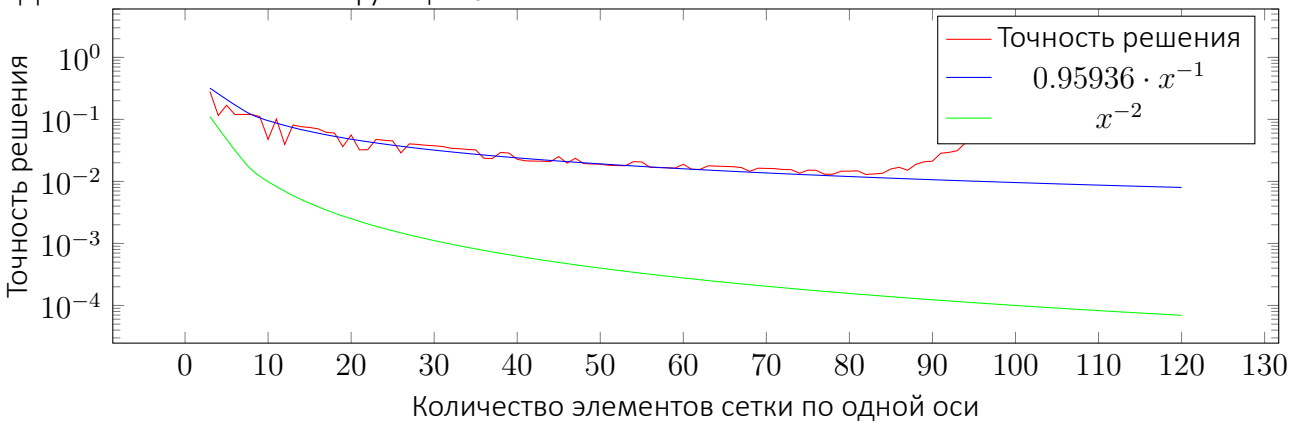
4.1 Точность для разных функций

Функция	Квадратная область, первые краевые условия	Ш-образная область, первые краевые условия	Ш-образная область, третьи краевые условия
$2x + y$	$1.59 \cdot 10^{-9}$	$1.52 \cdot 10^{-9}$	$4.21 \cdot 10^{-9}$
$3x^2 + y^2$	$3.3 \cdot 10^{-9}$	$3.41 \cdot 10^{-9}$	2.18
$x^3 + xy^2 + y^3$	$2.41 \cdot 10^{-9}$	$1.25 \cdot 10^{-9}$	1.64
$x^4 + y^4$	$8.67 \cdot 10^{-3}$	$6.17 \cdot 10^{-3}$	2.52
$x^5 + y^5 + 2xy$	$2.19 \cdot 10^{-2}$	$1.41 \cdot 10^{-2}$	3.86
e^{x+y}	$1.04 \cdot 10^{-3}$	$6.68 \cdot 10^{-4}$	1.61
$e^{x^2+y^2}$	$2.3 \cdot 10^{-2}$	$1.28 \cdot 10^{-2}$	5.67
$e^{x^3+yx^2}$	$9.16 \cdot 10^{-2}$	$5.61 \cdot 10^{-2}$	15.11
$sinx + cosy$	$2.4 \cdot 10^{-4}$	$1.77 \cdot 10^{-4}$	0.31
$sqrtx^2 + y^2$	$4.75 \cdot 10^{-3}$	$4.59 \cdot 10^{-3}$	1.03
$x^{1.2} + y^{1.5}$	$1.88 \cdot 10^{-2}$	$1.64 \cdot 10^{-2}$	NaN

4.2 Зависимость точности от размера сетки

Параметры остаются прежними, с небольшими изменениями:

● Количество элементов по обеим осям является переменной величиной.
● Для неполиномиальной функции $e^{x^3+yx^2}$.

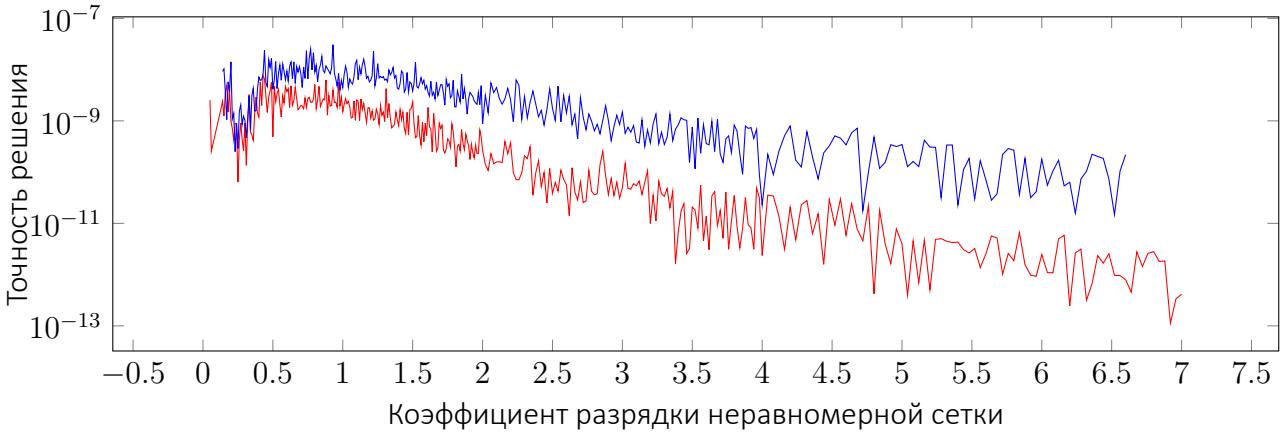


4.3 Зависимость точности от параметра разрядки неравномерной сетки

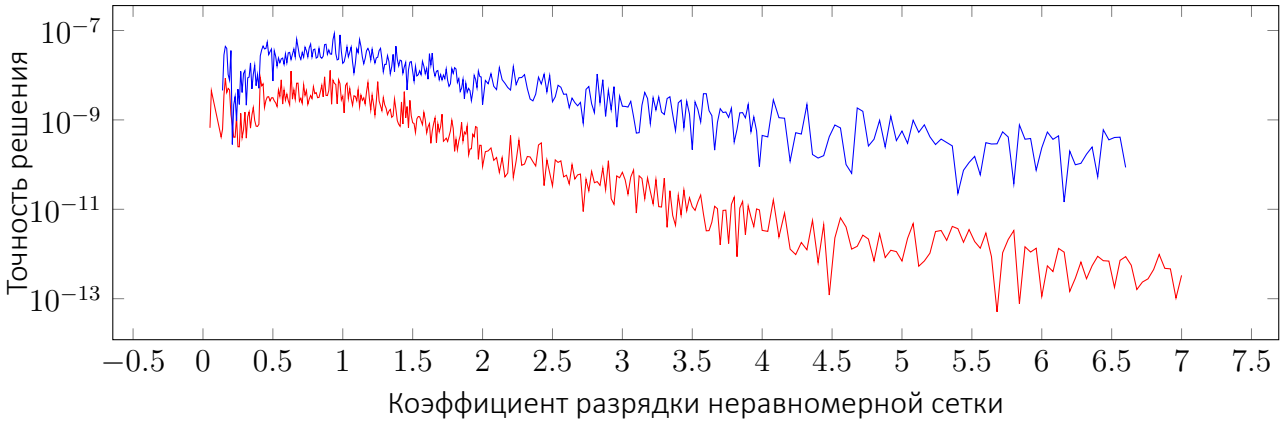
Параметры остаются прежними, с небольшими изменениями:

● Область сетки: $[0, 1] \times [0, 1]$ для красных графиков, и $[1, 2] \times [2, 3]$ для синих графиков.
● Сетка неравномерная.
● Коэффициент разрядки неравномерной сетки является переменной величиной.
Для коэффициента разрядки 1 сетка получается равномерной.

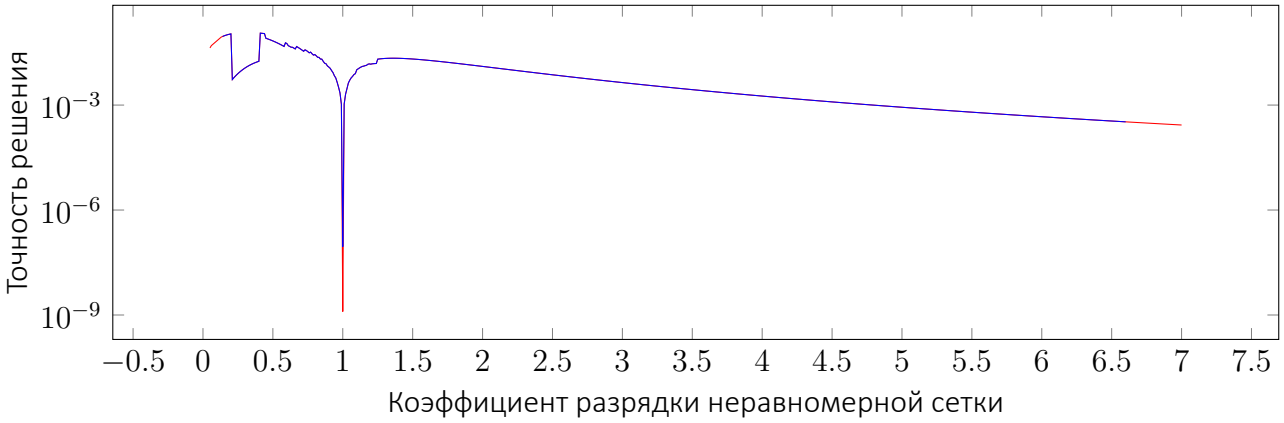
4.3.1 $2x + y$



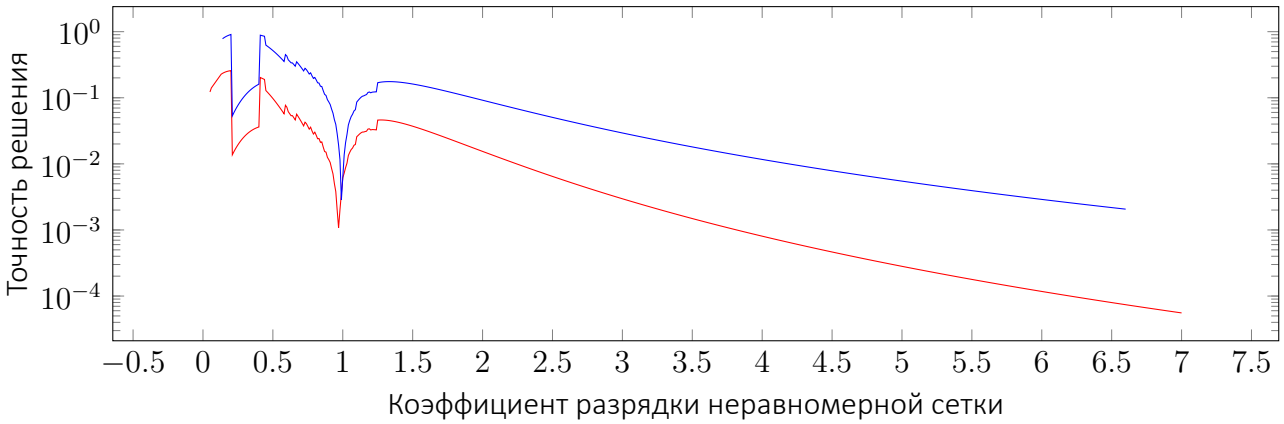
4.3.2 $3x^2 + y^2$



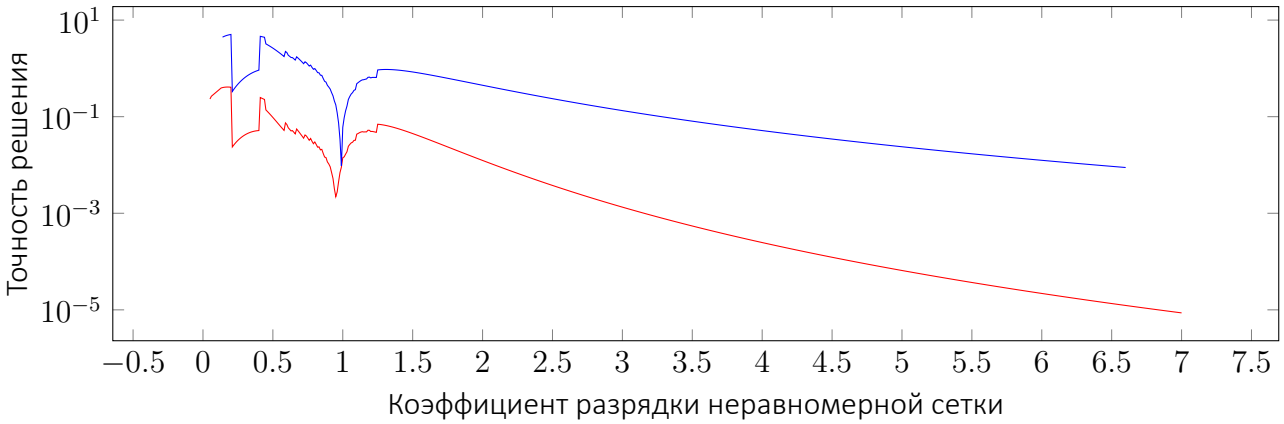
4.3.3 $x^3 + xy^2 + y^3$



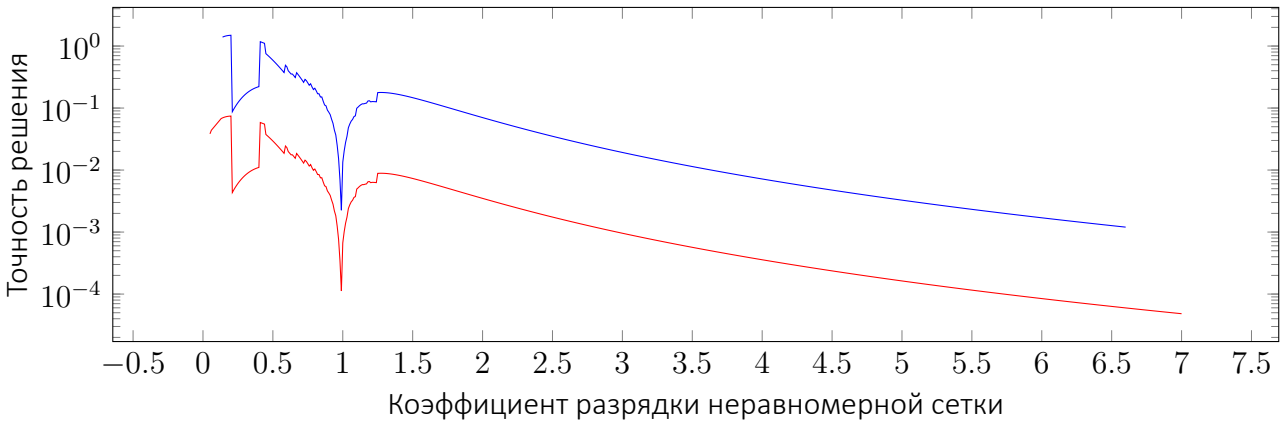
4.3.4 $x^4 + y^4$



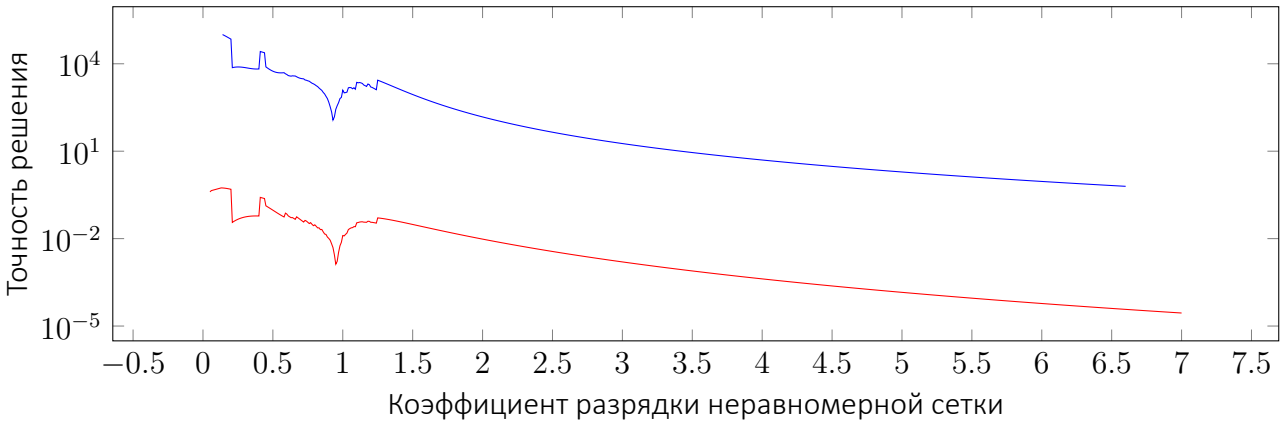
4.3.5 $x^5 + y^5 + 2xy$



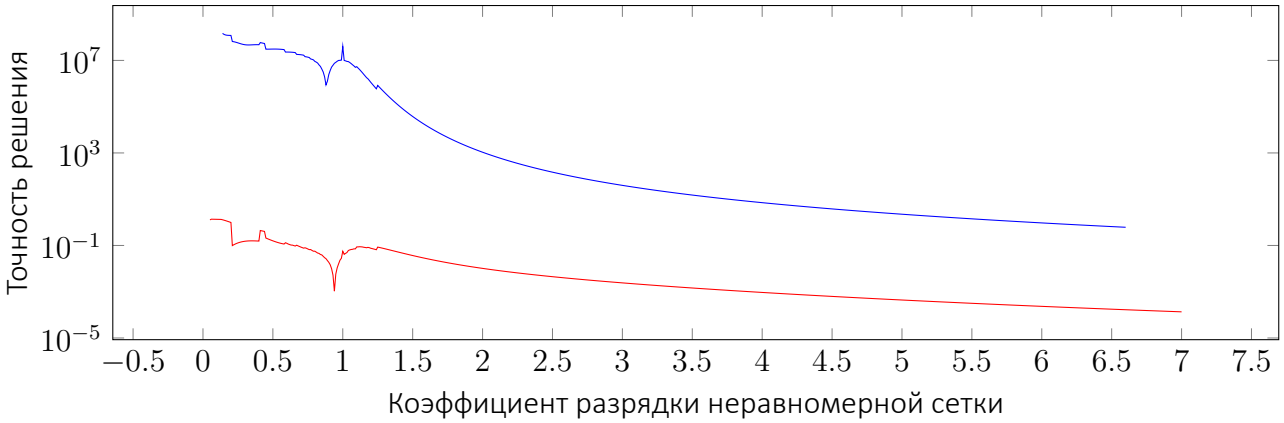
4.3.6 e^{x+y}



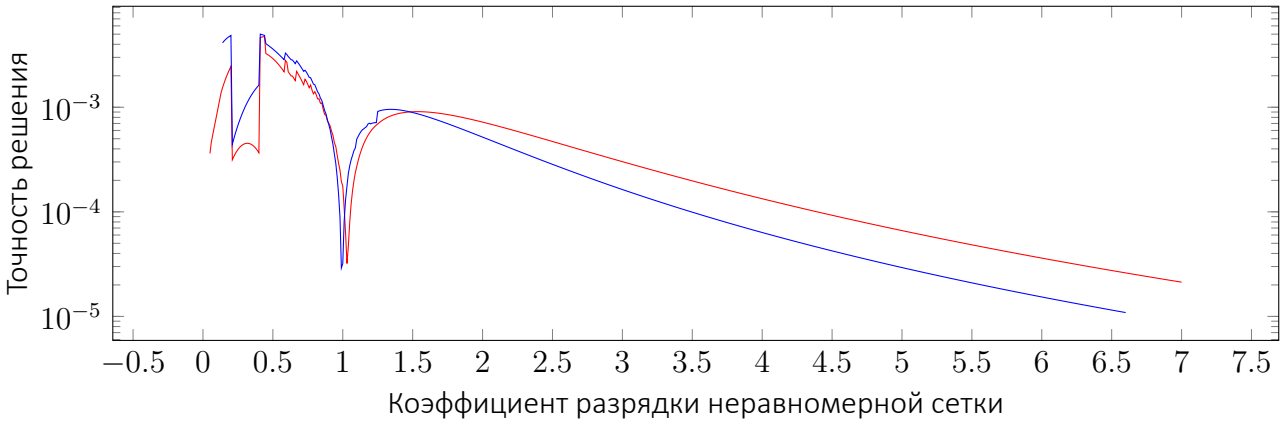
4.3.7 $e^{x^2+y^2}$



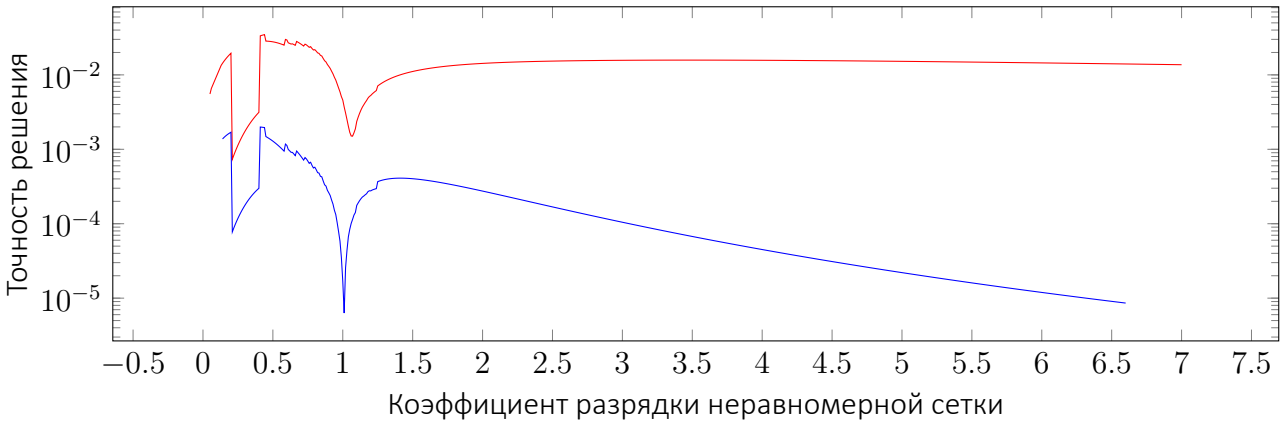
4.3.8 $e^{x^3+yx^2}$



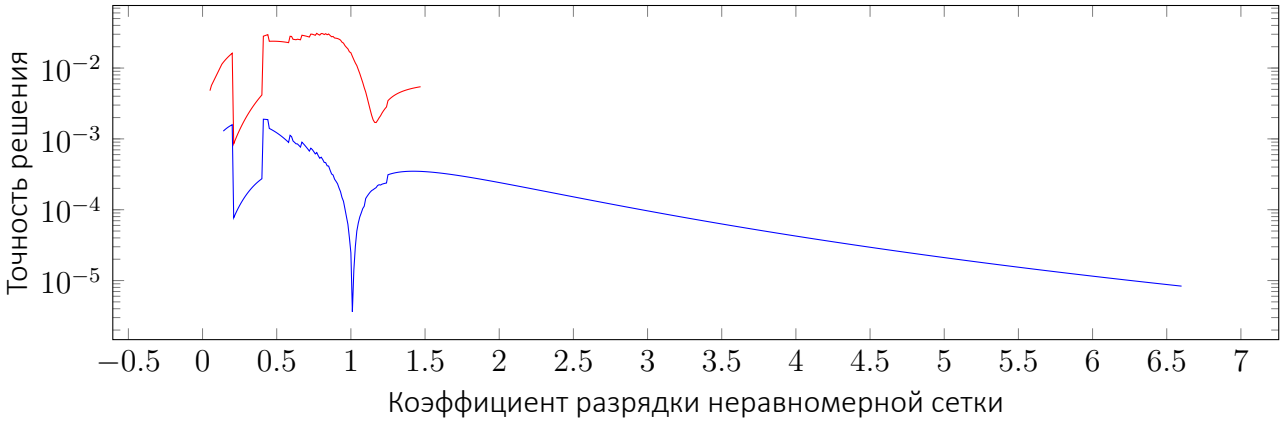
4.3.9 $\sin x + \cos y$



4.3.10 $\sqrt{x^2 + y^2}$



4.3.11 $x^{1.2} + y^{1.5}$



5 Выводы

● Порядок аппроксимации метода конечных разностей с первыми краевыми условиями равен 3, с третьими краевыми условиями равен 1.
● На неравномерных сетках с первыми краевыми условиями можно получить решение, стремящееся к точному, изменяя коэффициент разрядки.
● Согласно графику зависимости точности решения от размера сетки, можно вычислить, что порядок сходимости равен 1.
● Для неполиномиальных функций смещение области расчета может изменять точность решения.

6 Код программы

Для решения СЛАУ и хранения матрицы в диагональном формате был использован код из 2 лабораторной работы по Численным Методам.

6.1 Основной код

FILE **main.cpp**

```cpp
#include <iostream>
#include <functional>
#include <fstream>
#include <algorithm>
#include <cassert>
#include <diagonal.h>

typedef std::function<double(double)> Function1D;
typedef std::function<double(double, double)> Function2D;
typedef std::pair<Function2D, std::string> NamedFunction;

struct Cell
{
    int i;

    enum
    {
        FICTITIOUS, // Фиктивный узел
        INNER, // Нет краевых условий
        FIRST, // f(x, y) = value
        SECOND, // f'(x, y) = value
        THIRD, // f'(x, y) + c1 * f(x, y) + c2 = 0
    }
    conditionType; // краевое условие

    union {
        struct {
            double value;
        } first;
        struct {
            Cell* cell_around;
            double value;
            double h;
        } second;
        struct {
            Cell* cell_around;
            double value;
            double h;
            double c1;
        } third;
    } conditionValue;

    double value;
    double x, y;

    Cell *up, *down, *left, *right;
};

typedef std::vector<Cell> Cells;

class Field
{
public:
    Field(double startx, double starty, double sizex, double sizey) : sizex(sizex), sizey(sizey),
    ↪  startx(startx), starty(starty) {
    }

    virtual bool isPointInside(double x, double y) const = 0;

    double sizex, sizey;
    double startx, starty;
};

bool isInsideSquare(double x, double y, double sizex, double sizey, double startx, double starty) {
    x -= startx;
    y -= starty;
    return x >= 0 && x <= sizex && y >= 0 && y <= sizey;
}

class SquareField : public Field
{
public:
    SquareField(double startx, double starty, double sizex, double sizey) : Field(startx, starty,
    ↪  sizex, sizey) {
    }
    bool isPointInside(double x, double y) const {
        if (isInsideSquare(x, y, sizex, sizey, startx, starty))
            return true;
        else
            return false;
    }
};
```

```cpp

class ShField : public Field
{
public:
    ShField(double startx, double starty, double sizex, double sizey) : Field(startx, starty, sizex,
    ↪  sizey) {
    }
    bool isPointInside(double x, double y) const {
        if (isInsideSquare(x, y, sizex, sizey, startx, starty) &&
            !isInsideSquare(x, y, sizex * 0.2, sizey * 0.2, startx + sizex * 0.2, starty + sizey *
            ↪  0.8) &&
            !isInsideSquare(x, y, sizex * 0.2, sizey * 0.2, startx + sizex * 0.6, starty + sizey * 0.8
))
            return true;
        else
            return false;
    }
};

class Grid
{
public:
    virtual Cells makeCells(const Field& field,
                            double startx, double starty,
                            double sizex, double sizey,
                            int isizex, int isizey) = 0;
};

class NonUniformGrid : public Grid
{
public:
    NonUniformGrid(double c) : c(c) {}

    Cells makeCells(const Field& field,
                    double startx, double starty,
                    double sizex, double sizey,
                    int isizex, int isizey) {
        Cells cells;
        cells.reserve(isizex * isizey);
        // [строка][столбец]
        std::vector<std::vector<int>> grid(isizey, std::vector<int>(isizex, -1));

        double x = startx, y = starty;
        double hx, hy;
        if (std::fabs(c - 1) > 0.00001) {
            hx = sizex * (1.0-c)/(1-pow(c, isizex));
            hy = sizey * (1.0-c)/(1-pow(c, isizey));
        } else {
            // Это равномерная сетка
            hx = sizex/(isizex-1);
            hy = sizey/(isizey-1);
        }
        double oldhx = hx;
        int k = 0;
        for (int i = 0; i < isizey; ++i, y += hy, hy *= c) {
            x = startx;
            hx = oldhx;
            for (int j = 0; j < isizex; ++j, x += hx, hx *= c) {
                if (field.isPointInside(x, y)) {
                    grid[i][j] = k;
                    cells.push_back({k, Cell::INNER, 0, 0, x, y,
                                     nullptr,
                                     (i != 0 && grid[i-1][j] != -1) ? &cells[grid[i-1][j]] : nullptr,
                                     (j != 0 && grid[i][j-1] != -1) ? &cells[grid[i][j-1]] : nullptr,
                                     nullptr,
                                     });
                    k++;
                    if (i != 0 && grid[i-1][j] != -1) cells[grid[i-1][j]].up = &cells.back();
                    if (j != 0 && grid[i][j-1] != -1) cells[grid[i][j-1]].right = &cells.back();
                } else {
                    cells.push_back({k, Cell::FICTITIOUS, 0, 0, x, y, nullptr, nullptr, nullptr,
                    ↪  nullptr});
                    k++;
                }
            }
        }

        return cells;
    }
private:
    double c;
};

class UniformGrid : public Grid
{
public:
    UniformGrid() : grid(1.0) {
    }
```

```
    Cells makeCells(const Field& field,
                    double startx, double starty,
                    double sizex, double sizey,
                    int isizex, int isizey) {
        return grid.makeCells(field, startx, starty, sizex, sizey, isizex, isizey);
    }
private:
    NonUniformGrid grid;
};

Function2D calcFirstDerivativeX(const Function2D& f) {
    return [f] (double x, double y) -> double {
        const double h = 0.00001;
        return (-f(x+2*h, y)+8*f(x+h, y)-8*f(x-h, y)+f(x-2*h, y))/(12*h);
    };
}

Function2D calcFirstDerivativeY(const Function2D& f) {
    return [f] (double x, double y) -> double {
        const double h = 0.00001;
        return (-f(x, y+2*h)+8*f(x, y+h)-8*f(x, y-h)+f(x, y-2*h))/(12*h);
    };
}

Function2D calcSecondDerivativeX(const Function2D& f) {
    return [f] (double x, double y) -> double {
        const double h = 0.0001;
        return (-f(x+2*h, y) + 16*f(x+h, y) - 30*f(x, y) + 16*f(x-h, y) - f(x-2*h, y))/(12*h*h);
    };
}

Function2D calcSecondDerivativeY(const Function2D& f) {
    return [f] (double x, double y) -> double {
        const double h = 0.0001;
        return (-f(x, y+2*h) + 16*f(x, y+h) - 30*f(x, y) + 16*f(x, y-h) - f(x, y-2*h))/(12*h*h);
    };
}

Function2D calcLaplacian(const Function2D& f) {
    auto fxx = calcSecondDerivativeX(f);
    auto fyy = calcSecondDerivativeY(f);
    return [fxx, fyy] (double x, double y) -> double {
        return fxx(x, y) + fyy(x, y);
    };
}

void fillWithFunction(Cells& cells, const Function2D& f) {
    for (auto& i : cells) {
        i.value = f(i.x, i.y);
    }
}

void fillBoundaryConditions1(Cells& cells, const Function2D& f) {
    // Первые краевые условия
    for (auto& i : cells) {
        if (i.conditionType != Cell::FICTITIOUS && (i.up == nullptr || i.down == nullptr || i.left ==
        ↪  nullptr || i.right == nullptr)) {
            i.conditionType = Cell::FIRST;
            i.conditionValue.first.value = f(i.x, i.y);
        }
    }
}

void fillBoundaryConditions2(Cells& cells, const Function2D& f) {
    auto fx = calcFirstDerivativeX(f);
    auto fy = calcFirstDerivativeY(f);
    int isFirst = 2;
    for (auto& i : cells) {
        if (i.conditionType != Cell::FICTITIOUS && (i.up == nullptr || i.down == nullptr || i.left ==
        ↪  nullptr || i.right == nullptr)) {
            if (isFirst || i.left == nullptr && i.right == nullptr || i.up == nullptr && i.down ==
            ↪  nullptr) {
                // Тут уже ничего не сделаешь, потому что это палка, у которой нет соседей, чтобы
                ↪  можно было сделать краевые условия с производной
                i.conditionType = Cell::FIRST;
                i.conditionValue.first.value = f(i.x, i.y);
                isFirst--;
            } else {
                i.conditionType = Cell::SECOND;
                double fxv = fx(i.x, i.y);
                double fyv = fy(i.x, i.y);

                Cell* cell_around;
                double& result = i.conditionValue.second.value;
                if (i.left == nullptr) {
                    result = -fxv;
                    cell_around = i.right;
```

```
                } else if (i.right == nullptr) {
                    result = fxv;
                    cell_around = i.left;
                } else if (i.up == nullptr) {
                    result = fyv;
                    cell_around = i.down;
                } else if (i.down == nullptr) {
                    result = -fyv;
                    cell_around = i.up;
                } else {
                    throw std::exception("This is inner cell!");
                }
                i.conditionValue.second.cell_around = cell_around;

                double hx = std::fabs(cell_around->x - i.x);
                double hy = std::fabs(cell_around->y - i.y);
                double h = std::max(hx, hy);
                i.conditionValue.second.h = h;
            }
        }
    }
}

void fillBoundaryConditions3(Cells& cells, const Function2D& f, double c1) {
    auto fx = calcFirstDerivativeX(f);
    auto fy = calcFirstDerivativeY(f);
    for (auto& i : cells) {
        if (i.conditionType != Cell::FICTITIOUS && (i.up == nullptr || i.down == nullptr || i.left ==
        ↪  nullptr || i.right == nullptr)) {
            if (i.left == nullptr && i.right == nullptr || i.up == nullptr && i.down == nullptr) {
                // Тут уже ничего не сделаешь, потому что это палка, у которой нет соседей, чтобы
                ↪  можно было сделать краевые условия с производной
                i.conditionType = Cell::FIRST;
                i.conditionValue.first.value = f(i.x, i.y);
            } else {
                i.conditionType = Cell::THIRD;
                double fxv = fx(i.x, i.y);
                double fyv = fy(i.x, i.y);

                Cell* cell_around;
                double& result = i.conditionValue.third.value;
                if (i.left == nullptr) {
                    result = -fxv;
                    cell_around = i.right;
                } else if (i.right == nullptr) {
                    result = fxv;
                    cell_around = i.left;
                } else if (i.up == nullptr) {
                    result = fyv;
                    cell_around = i.down;
                } else if (i.down == nullptr) {
                    result = -fyv;
                    cell_around = i.up;
                } else {
                    throw std::exception("This is inner cell!");
                }
                result += f(i.x, i.y)*c1;
                i.conditionValue.third.cell_around = cell_around;

                double hx = std::fabs(cell_around->x - i.x);
                double hy = std::fabs(cell_around->y - i.y);
                double h = std::max(hx, hy);
                i.conditionValue.third.h = h;
                i.conditionValue.third.c1 = c1;
            }
        }
    }
}

double calcDifference(const Cells& a, const Cells& b) {
    if (a.size() != b.size())
        return -1;
    double sum = 0;
    for (int i = 0; i < a.size(); ++i) {
        if (a[i].conditionType != Cell::FICTITIOUS) {
            if (a[i].x != b[i].x || a[i].y != b[i].y)
                return -1;
            sum += (a[i].value - b[i].value)*(a[i].value - b[i].value);
        }
    }
    return std::sqrt(sum);
}

std::pair<MatrixDiagonal, Vector> makeSLAE(const Cells& cells, const Function2D& rightPart) {
    int n = sqrt(cells.size());
    assert(n*n == cells.size());
    MatrixDiagonal A(cells.size(), { 0, 1, -1, n, -n });
    Vector b(cells.size(), 0);
```

```
    struct PushType
    {
        int i;
        double value;
    };

    matrix_diagonal_line_iterator it(A.dimension(), A.getFormat(), false);
    auto pushLine = [&](const std::vector<PushType>& p) {
        for (; !it.isLineEnd(); ++it) {
            for (auto& i : p) {
                if (i.i == it.j)
                    A.begin(it.dn)[it.di] = i.value;
            }
        }
        ++it;
    };

    for (int i = 0; i < cells.size(); ++i) {
        const auto& cell = cells[i];
        switch (cell.conditionType) {
            case Cell::FICTITIOUS: {
                pushLine({{cell.i, 1 }});
                b(i) = 0;
            } break;
            case Cell::INNER: {
                double uph = std::fabs(cell.up->y - cell.y);
                double downh = std::fabs(cell.down->y - cell.y);
                double lefth = std::fabs(cell.left->x - cell.x);
                double righth = std::fabs(cell.right->x - cell.x);
                pushLine({
                    {cell.left->i, 2.0/(lefth*(righth + lefth))},
                    {cell.down->i, 2.0/(downh*(uph + downh))},
                    {cell.right->i, 2.0/(righth*(righth + lefth))},
                    {cell.up->i, 2.0/(uph*(uph + downh))},
                    {cell.i, -2.0/(uph*downh)-2.0/(lefth*righth)},
                });

                b(i) = rightPart(cell.x, cell.y);
            } break;
            case Cell::FIRST: {
                pushLine({{cell.i, 1}});
                b(i) = cell.conditionValue.first.value;
            } break;
            case Cell::SECOND: {
                double h = cell.conditionValue.second.h;
                pushLine({
                    {cell.i, 1.0/h},
                    {cell.conditionValue.second.cell_around->i, -1.0/h},
                });
                b(i) = cell.conditionValue.second.value;
            } break;
            case Cell::THIRD: {
                double h = cell.conditionValue.third.h;
                pushLine({
                    {cell.i, 1.0/h + cell.conditionValue.third.c1},
                    {cell.conditionValue.third.cell_around->i, -1.0/h},
                });
                b(i) = cell.conditionValue.second.value;
            } break;
        }
    }

    return {A, b};
}

Vector solveSLAE(std::pair<MatrixDiagonal, Vector>& slae) {
    MatrixDiagonal& A = slae.first;
    Vector& b = slae.second;
    Vector result;
    SolverSLAE_Iterative solver;
    solver.epsilon = 1e-12;
    solver.isLog = false;
    solver.maxIterations = 3000;
    solver.start = b;
    solver.w = 1.4;
    auto returned = solver.seidel(A, b, result);
    //std::cout << returned.relativeResidual << std::endl;
    return result;
}

void setCells(Cells& cells, const Vector& answer) {
    for (int i = 0; i < cells.size(); i++)
        cells[i].value = answer(i);
}

void make_table_nonuniform_grid(const std::string& name, const Field& field, const Function2D& f, int
↪  size, int boundaryCondition) {
```

```
    auto rightPart = calcLaplacian(f);
    std::ofstream fout(name);
    fout << "c\tnorm" << std::endl;
    for (int i = 1; i <= 700;) {
        NonUniformGrid grid(i/100.0);
        auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizex,
        ↪  field.sizey, size, size);
        auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizex,
        ↪  field.sizey, size, size);
        fillWithFunction(cells_answer, f);
        if (boundaryCondition == 1) {
            fillBoundaryConditions1(cells_question, f);
        } else if (boundaryCondition == 2) {
            fillBoundaryConditions2(cells_question, f);
        } else {
            fillBoundaryConditions3(cells_question, f, 1);
        }
        auto slae = makeSLAE(cells_question, rightPart);
        auto answer = solveSLAE(slae);
        setCells(cells_question, answer);
        double difference = calcDifference(cells_answer, cells_question);
        if (difference == difference) {
            fout << i / 100.0 << "\t" << difference << std::endl;
        }

        std::cout << "\r" << i;

        if (i < 200) {
            i++;
        } else {
            if (i < 400) {
                i += 2;
            } else {
                if (i < 1000) {
                    i += 4;
                } else {
                    i += 10;
                }
            }
        }
    }
    fout.close();
}

void make_table_size(const std::string& name, const Field& field, const Function2D& f, int
↪  boundaryCondition) {
    auto rightPart = calcLaplacian(f);
    std::ofstream fout(name);
    fout << "size\tnorm" << std::endl;
    for (int i = 3; i <= 120; i++) {
        UniformGrid grid;
        auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizex,
        ↪  field.sizey, i, i);
        auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizex,
        ↪  field.sizey, i, i);
        fillWithFunction(cells_answer, f);
        if (boundaryCondition == 1) {
            fillBoundaryConditions1(cells_question, f);
        } else if (boundaryCondition == 2) {
            fillBoundaryConditions2(cells_question, f);
        } else {
            fillBoundaryConditions3(cells_question, f, 1);
        }
        auto slae = makeSLAE(cells_question, rightPart);
        auto answer = solveSLAE(slae);
        setCells(cells_question, answer);
        double difference = calcDifference(cells_answer, cells_question);
        if (difference == difference)
            fout << i << "\t" << difference << std::endl;

        std::cout << "\r" << i;
    }
    fout.close();
}

void make_table_conditions(const std::string& name, const Field& field, const
↪  std::vector<NamedFunction>& namedFunctions) {
    std::ofstream fout(name);
    fout << "func\t1square\t1sh\t3sh" << std::endl;

    SquareField field1(field.startx, field.starty, field.sizex, field.sizey);
    ShField field2(field.startx, field.starty, field.sizex, field.sizey);
    UniformGrid grid;
    int size = 20;

    for (auto& i : namedFunctions) {
        fout << i.second << "\t";
        auto& f = i.first;
```

```
        auto rightPart = calcLaplacian(f);

        {
                auto& field = field1;
            auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizex,
            ↪  field.sizey, size, size);
            auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizex,
            ↪  field.sizey, size, size);
            fillWithFunction(cells_answer, f);
                fillBoundaryConditions1(cells_question, f);
            auto slae = makeSLAE(cells_question, rightPart);
            auto answer = solveSLAE(slae);
            setCells(cells_question, answer);
            double difference = calcDifference(cells_answer, cells_question);

            fout << difference << "\t";
        }

        {
                auto& field = field2;
            auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizex,
            ↪  field.sizey, size, size);
            auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizex,
            ↪  field.sizey, size, size);
            fillWithFunction(cells_answer, f);
                fillBoundaryConditions1(cells_question, f);
            auto slae = makeSLAE(cells_question, rightPart);
            auto answer = solveSLAE(slae);
            setCells(cells_question, answer);
            double difference = calcDifference(cells_answer, cells_question);

            fout << difference << "\t";
        }

        {
                auto& field = field2;
            auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizex,
            ↪  field.sizey, size, size);
            auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizex,
            ↪  field.sizey, size, size);
            fillWithFunction(cells_answer, f);
                fillBoundaryConditions3(cells_question, f, 1);
            auto slae = makeSLAE(cells_question, rightPart);
            auto answer = solveSLAE(slae);
            setCells(cells_question, answer);
            double difference = calcDifference(cells_answer, cells_question);

            fout << difference << std::endl;
        }
    }
    fout.close();
}

int main() {
    std::vector<NamedFunction> namedFunctions;
    namedFunctions.push_back({[] (double x, double y) -> double { return 2*x+y; }, "{$2x+y$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return 3*x*x + y*y + x; },
    ↪  "{$3x^2+y^2$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return x*x*x + x*y*y + y*y*y; },
    ↪  "{$x^3+xy^2+y^3$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return x*x*x*x + y*y*y*y; },
    ↪  "{$x^4+y^4$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return x*x*x*x*x + y*y*y*y*y +
    ↪  2*x*y; }, "{$x^5+y^5+2xy$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return exp(x+y); }, "{$e^{x+y}$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return exp(x*x+y*y); },
    ↪  "{$e^{x^2+y^2}$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return exp(x*x*x+x*x*y); },
    ↪  "{$e^{x^3+yx^2}$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return sin(x)+cos(y); }, "{$\sin x +
    ↪  \cos y$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return sqrt(x*x+y*y); },
    ↪  "{$\sqrt{x^2+y^2}$}"});
    namedFunctions.push_back({[] (double x, double y) -> double { return pow(x, 1.2) + pow(y, 1.5); },
    ↪  "{$x^{1.2}+y^{1.5}$}"});

    auto make_tables = [&](std::string add, const Field& field) {
        std::cout << "Make tables for " << add << std::endl;

        std::cout << "\rWrite condition table" << std::endl;
        make_table_conditions(add + "conditions.txt", field, namedFunctions);

        std::cout << "\rWrite size table" << std::endl;
        //make_table_size(add + "size.txt", field, namedFunctions[7].first, 1);

        std::cout << "\rWrite nonuniform grid tables" << std::endl;
```

```
        for (int i = 0; i < namedFunctions.size(); i++) {
            std::cout << "\r" << i << "                              " << std::endl;
            make_table_nonuniform_grid(std::string(add + "non_uniform_grid_") + std::to_string(i) +
            ↪  std::string(".txt"), field, namedFunctions[i].first, 20, 1);
        }
    };

    make_tables("0_0_", ShField(0, 0, 1, 1));
    make_tables("1_2_", ShField(1, 2, 1, 1));
}
```

6.2 Код из другой лабораторной работы

6.2.1 Заголовочные файлы

diagonal.h

```
#pragma once

#include <string>
#include <vector>
#include <iostream>
#include <map>
#include <functional>
#include "../1/common.h"
#include "../1/vector.h"
#include "../1/matrix.h"

class MatrixDiagonal;
class matrix_diagonal_iterator;
class SolverSLAE_Iterative;

//---------------------------------------------------------------------------
/** Класс для вычислений различных параметров с диагональными матрицами. */
class Diagonal
{
public:
    int n;

    //---------------------------------------------------------------------------
    Diagonal(int n);

    int calcDiagonalsCount(void);
    int calcMinDiagonal(void);
    int calcMaxDiagonal(void);
    int calcDiagonalSize(int d);

    bool isLineIntersectDiagonal(int line, int d);
    bool isRowIntersectDiagonal(int row, int d);

    //---------------------------------------------------------------------------
    /*
        R - Row - столбец
        L - Line - строка
        P - Pos - номер элемента в диагонали
        D - Diag - формат диагонали
    */

    int calcLine_byDP(int d, int pos);
    int calcRow_byDP(int d, int pos);

    int calcDiag_byLR(int line, int row);
    int calcPos_byLR(int line, int row);

    int calcPos_byDL(int d, int line);
    int calcPos_byDR(int d, int row);

    int calcRow_byDL(int d, int line);
    int calcLine_byDR(int d, int row);
};

//---------------------------------------------------------------------------
/** Матрица в диагональном формате. */
/** 0-я диагональ всегда главная диагональ. */
class MatrixDiagonal
{
public:
    typedef std::vector<real>::iterator iterator;
    typedef std::vector<real>::const_iterator const_iterator;

    //---------------------------------------------------------------------------
    MatrixDiagonal();
```

```
    MatrixDiagonal(int n, std::vector<int> format); // format[0] must be 0, because it's main diagonal
    MatrixDiagonal(const Matrix& a);

    void toDenseMatrix(Matrix& dense) const;
    void resize(int n, std::vector<int> format); // format[0] must be 0, because it's main diagonal

    void save(std::ostream& out) const;
    void load(std::istream& in);

    //--------------------------------------------------------------------------
    int dimension(void) const;
    int getDiagonalsCount(void) const;
    int getDiagonalSize(int diagNo) const;
    int getDiagonalPos(int diagNo) const;

    std::vector<int> getFormat(void) const;

    //--------------------------------------------------------------------------
    matrix_diagonal_iterator posBegin(int diagNo) const;
    matrix_diagonal_iterator posEnd(int diagNo) const;

    iterator begin(int diagNo);
    const_iterator begin(int diagNo) const;

    iterator end(int diagNo);
    const_iterator end(int diagNo) const;

private:
    std::vector<std::vector<real>> di;
    std::vector<int> fi;
    int n;
    Diagonal dc;
};

std::vector<int> makeSevenDiagonalFormat(int n, int m, int k);
std::vector<int> generateRandomFormat(int n, int diagonalsCount);

void generateDiagonalMatrix(
    int n,
    int min, int max,
    std::vector<int> format,
    MatrixDiagonal& result
);

void generateDiagonallyDominantMatrix(
    int n,
    std::vector<int> format,
    bool isNegative,
    MatrixDiagonal& result
);

bool mul(const MatrixDiagonal& a, const Vector& x, Vector& y);

//--------------------------------------------------------------------------
/** Матричный "итератор" для движения по диагонали. */
class matrix_diagonal_iterator
{
public:
    matrix_diagonal_iterator(int n, int d, bool isEnd);

    matrix_diagonal_iterator& operator++();
    matrix_diagonal_iterator  operator++(int);

    bool operator==(const matrix_diagonal_iterator& b) const;
    bool operator!=(const matrix_diagonal_iterator& b) const;

    matrix_diagonal_iterator& operator+=(const ptrdiff_t& movement);

    int i, j;
};

/** Матричный "итератор" для движения по строке между различными диагоналями. Может обрабатывать как
↪  всю строку, так и только нижний треугольник. */
class matrix_diagonal_line_iterator
{
public:
    matrix_diagonal_line_iterator(int n, std::vector<int> format, bool isOnlyLowTriangle);

    matrix_diagonal_line_iterator& operator++();
    matrix_diagonal_line_iterator  operator++(int);

    bool isLineEnd(void) const;
    bool isEnd(void) const;

    int i, j; // i - текущая строка,  j - текущий столбец
    int d, dn, di; // d - формат текущей диагонали, d - номер текущей диагонали, di - номер текущего
    ↪  элемента в диагонали
private:
    std::map<int, int> m_map;
```

```
    std::vector<int> m_sorted_format;

    int line, start, end, pos;
    Diagonal dc;

    bool m_isLineEnd;
    bool m_isEnd;

    void calcPos(void);
};

//---------------------------------------------------------------------------
/** Класс итеративного решателя СЛАУ для диагональной матрицы. */
struct IterationsResult
{
    int iterations;
    double relativeResidual;
};

class SolverSLAE_Iterative
{
public:
    SolverSLAE_Iterative();

    void save(std::ostream& out) const;
    void load(std::istream& in);

    IterationsResult jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x) const;
    IterationsResult seidel(const MatrixDiagonal& a, const Vector& y, Vector& x) const;

    double             w;
    bool             isLog;
    std::ostream&     log;
    Vector             start;
    double             epsilon;
    int             maxIterations;

private:
    mutable Vector x1;

    // До итерации: x - текущее решение. После итерации x - следующее решение.
    void iteration_jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x) const;
    void iteration_seidel(const MatrixDiagonal& a, const Vector& y, Vector& x) const;

    typedef std::function<void(const SolverSLAE_Iterative*, const MatrixDiagonal&, const Vector&,
    ↪  Vector&)> step_function;

    IterationsResult iteration_process(
        const MatrixDiagonal& a,
        const Vector& y,
        Vector& x,
        step_function step
    ) const;
};
```

FILE **vector.h**

```
#pragma once

#include <string>
#include <vector>
#include "matrix.h"
#include "common.h"

//---------------------------------------------------------------------------
class Vector
{
public:
    Vector();
    Vector(int size, real fill = 0);
    Vector(const Matrix& a);

    void loadFromFile(std::string fileName);
    void saveToFile(std::string fileName) const;

    void save(std::ostream& out) const;
    void load(std::istream& in);

    void generate(int n, int min, int max);
    void generate(int n);

    //---------------------------------------------------------------------------
    void resize(int n, real fill = 0);

    void negate(void);

```

```
    void zero(void);

    void toDenseMatrix(Matrix& dense, bool isVertical) const;

    //-------------------------------------------------------------------------
    int size(void) const;

    real& operator()(int i);
    const real& operator()(int i) const;

private:
    std::vector<real> mas;
};

//-------------------------------------------------------------------------
sumreal sumAllElementsAbs(const Vector& a);
bool sum(const Vector& a, const Vector& b, Vector& result);
bool mul(const Matrix& a, const Vector& b, Vector& result);
real calcNorm(const Vector& a);
```

FILE **common.h**

```
#pragma once

#ifdef ALL_FLOAT
typedef float real;
typedef float sumreal;
#endif

#ifdef ALL_DOUBLE
typedef double real;
typedef double sumreal;
#endif

#ifdef ALL_FLOAT_WITH_DOUBLE
typedef float real;
typedef double sumreal;
#endif

#ifndef ALL_FLOAT
#ifndef ALL_DOUBLE
#ifndef ALL_FLOAT_WITH_DOUBLE
#error ”Type isn't defined”
typedef double real;
typedef double sumreal;
#endif
#endif
#endif


bool isNear(double a, double b);
double random(void);
int intRandom(int min, int max);
```

FILE **matrix.h**

```
#pragma once

#include <string>
#include <vector>
#include ”common.h”

//-------------------------------------------------------------------------
class Matrix
{
public:
    Matrix(int n = 0, int m = 0, real fill = 0); // n - количество столбцов, m - количество строк
    void loadFromFile(std::string fileName);
    void saveToFile(std::string fileName) const;

    void save(std::ostream& out) const;
    void load(std::istream& in);

    void getFromVector(int n, int m, const std::vector<real>& data);

    void resize(int n, int m, real fill = 0);
    void negate(void);

    bool isSymmetric(void) const;
    bool isLowerTriangular(void) const;
    bool isUpperTriangular(void) const;
    bool isDiagonal(void) const;
    bool isDiagonalIdentity(void) const;
    bool isDegenerate(void) const;

    real& operator()(int i, int j);
```

```
    const real& operator()(int i, int j) const;

    int width(void) const;
    int height(void) const;

private:
    std::vector<std::vector<real>> m_matrix;
    int m_n, m_m;
};

//---------------------------------------------------------------------------
void generateSparseSymmetricMatrix(
    int n,
    int min, int max,
    real percent,
    Matrix& result
);

void generateLMatrix(
    int n,
    int min, int max,
    real percent,
    Matrix& result
);

void generateDiagonalMatrix(
    int n,
    int min, int max,
    Matrix& result
);

void generateVector(
    int n,
    int min, int max,
    Matrix& result
);

void generateVector(
    int n,
    Matrix& result
);

void generateGilbertMatrix(int n, Matrix& result);

void generateTestMatrix(int n, int profileSize, Matrix& result);

//---------------------------------------------------------------------------
bool mul(const Matrix& a, const Matrix& b, Matrix& result);
bool sum(const Matrix& a, const Matrix& b, Matrix& result);

sumreal sumAllElementsAbs(const Matrix& a);

bool transpose(Matrix& a);

bool calcLDL(const Matrix& a, Matrix& l, Matrix& d);
bool calcLUsq(const Matrix& a, Matrix& l, Matrix& u);
bool calcLUsq_partial(const Matrix& a, Matrix& l, Matrix& u);
bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x);
bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x);
bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x);
bool solveSLAE_by_LDL(const Matrix& a, const Matrix& y, Matrix& x);

bool solveSLAE_byGaussMethod(const Matrix& a, const Matrix& y, Matrix& x);
```

6.2.2 Файлы исходного кода

FILE **diagonal.cpp**

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include ”diagonal.h”

//---------------------------------------------------------------------------
Diagonal::Diagonal(int n) : n(n) {
}

//---------------------------------------------------------------------------
int Diagonal::calcDiagonalsCount(void) {
    return 2 * n - 1;
}

```

```
//---------------------------------------------------------------------------
int Diagonal::calcMinDiagonal(void) {
    return -(n-1);
}

//---------------------------------------------------------------------------
int Diagonal::calcMaxDiagonal(void) {
    return n - 1;
}

//---------------------------------------------------------------------------
int Diagonal::calcDiagonalSize(int d) {
    return n - std::abs(d);
}

//---------------------------------------------------------------------------
bool Diagonal::isLineIntersectDiagonal(int line, int d) {
    if (d <= 0)
        return (line+d >= 0);

    if (d > 0)
        return (line < calcDiagonalSize(d));
}

//---------------------------------------------------------------------------
bool Diagonal::isRowIntersectDiagonal(int row, int d) {
    return isLineIntersectDiagonal(row, -d);
}

//---------------------------------------------------------------------------
int Diagonal::calcLine_byDP(int d, int pos) {
    if (d <= 0)
        return -d + pos;

    if (d > 0)
        return pos;
}

//---------------------------------------------------------------------------
int Diagonal::calcRow_byDP(int d, int pos) {
    if (d <= 0)
        return pos;

    if (d > 0)
        return pos + d;
}

//---------------------------------------------------------------------------
int Diagonal::calcDiag_byLR(int line, int row) {
    return row - line;
}

//---------------------------------------------------------------------------
int Diagonal::calcPos_byLR(int line, int row) {
    return calcPos_byDL(calcDiag_byLR(line, row), line);
}

//---------------------------------------------------------------------------
int Diagonal::calcPos_byDL(int d, int line) {
    if (d <= 0)
        return line+d;

    if (d > 0)
        return line;
}

//---------------------------------------------------------------------------
int Diagonal::calcPos_byDR(int d, int row) {
    return calcPos_byDL(d, calcLine_byDR(d, row));
}

//---------------------------------------------------------------------------
int Diagonal::calcRow_byDL(int d, int line) {
    return line+d;
}

//---------------------------------------------------------------------------
int Diagonal::calcLine_byDR(int d, int row) {
    return calcRow_byDL(-d, row);
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
MatrixDiagonal::MatrixDiagonal() : n(0), dc(n) {
}

```

```cpp
//---------------------------------------------------------------------------
MatrixDiagonal::MatrixDiagonal(int n, std::vector<int> format) : dc(n) {
    resize(n, format);
}

//---------------------------------------------------------------------------
MatrixDiagonal::MatrixDiagonal(const Matrix& a) : dc(n) {
    if (a.width() != a.height())
        throw std::exception();

    n = a.width();
    dc.n = n;

    // Определяем формат
    std::vector<int> format;
    format.clear();
    format.push_back(0);
    for (int i = dc.calcMinDiagonal(); i <= dc.calcMaxDiagonal(); ++i)
        if (i != 0) {
            auto mit = matrix_diagonal_iterator(n, i, false);
            auto mite = matrix_diagonal_iterator(n, i, true);
            for (; mit != mite; ++mit) {
                if (a(mit.i, mit.j) != 0) {
                    format.push_back(i);
                    break;
                }
            }
        }

    // Создаем формат
    resize(n, format);

    // Обходим массив и записываем элементы
    for (int i = 0; i < getDiagonalsCount(); ++i) {
        auto mit = posBegin(i);
        for (auto it = begin(i); it != end(i); ++it, ++mit)
            *it = a(mit.i, mit.j);
    }
}

//---------------------------------------------------------------------------
void MatrixDiagonal::toDenseMatrix(Matrix& dense) const {
    dense.resize(n, n, 0);

    // Обходим массив и записываем элементы
    for (int i = 0; i < getDiagonalsCount(); ++i) {
        auto mit = posBegin(i);
        for (auto it = begin(i); it != end(i); ++it, ++mit)
            dense(mit.i, mit.j) = *it;
    }
}

//---------------------------------------------------------------------------
void MatrixDiagonal::resize(int n1, std::vector<int> format) {
    if (format[0] != 0)
        throw std::exception();

    dc.n = n1;
    n = n1;
    fi = format;

    di.clear();
    for (const auto& i : format)
        di.push_back(std::vector<real>(dc.calcDiagonalSize(i), 0));
}

//---------------------------------------------------------------------------
void MatrixDiagonal::save(std::ostream& out) const {
    out << n << " " << fi.size() << std::endl;
    for (const auto& i : fi)
        out << i << " ";
    out << std::endl;

    for (int i = 0; i < getDiagonalsCount(); ++i) {
        for (auto j = begin(i); j != end(i); ++j) {
            out << (*j) << " ";
        }
        out << std::endl;
    }
    out << std::endl;
}

//---------------------------------------------------------------------------
void MatrixDiagonal::load(std::istream& in) {
    int n, m;
    in >> n >> m;
    std::vector<int> format(m, 0);
    for (int i = 0; i < m; ++i)
        in >> format[i];
```

```
    resize(n, format);
    for (int i = 0; i < getDiagonalsCount(); ++i) {
        for (auto j = begin(i); j != end(i); ++j) {
            in >> (*j);
        }
    }
}

//---------------------------------------------------------------------------
int MatrixDiagonal::dimension(void) const {
    return n;
}

//---------------------------------------------------------------------------
int MatrixDiagonal::getDiagonalsCount(void) const {
    return di.size();
}

//---------------------------------------------------------------------------
int MatrixDiagonal::getDiagonalSize(int diagNo) const {
    return di[diagNo].size();
}

//---------------------------------------------------------------------------
int MatrixDiagonal::getDiagonalPos(int diagNo) const {
    return fi[diagNo];
}

//---------------------------------------------------------------------------
std::vector<int> MatrixDiagonal::getFormat(void) const {
    return fi;
}

//---------------------------------------------------------------------------
matrix_diagonal_iterator MatrixDiagonal::posBegin(int diagNo) const {
    return matrix_diagonal_iterator(n, fi[diagNo], false);
}

//---------------------------------------------------------------------------
matrix_diagonal_iterator MatrixDiagonal::posEnd(int diagNo) const {
    return matrix_diagonal_iterator(n, fi[diagNo], true);
}

//---------------------------------------------------------------------------
MatrixDiagonal::iterator MatrixDiagonal::begin(int diagNo) {
    return di[diagNo].begin();
}

//---------------------------------------------------------------------------
MatrixDiagonal::const_iterator MatrixDiagonal::begin(int diagNo) const {
    return di[diagNo].begin();
}

//---------------------------------------------------------------------------
MatrixDiagonal::iterator MatrixDiagonal::end(int diagNo) {
    return di[diagNo].end();
}

//---------------------------------------------------------------------------
MatrixDiagonal::const_iterator MatrixDiagonal::end(int diagNo) const {
    return di[diagNo].end();
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
matrix_diagonal_iterator::matrix_diagonal_iterator(int n, int d, bool isEnd) {
    Diagonal dc(n);
    if (isEnd) {
        i = dc.calcLine_byDP(d, dc.calcDiagonalSize(d));
        j = dc.calcRow_byDP(d, dc.calcDiagonalSize(d));
    } else {
        i = dc.calcLine_byDP(d, 0);
        j = dc.calcRow_byDP(d, 0);
    }
}

//---------------------------------------------------------------------------
matrix_diagonal_iterator& matrix_diagonal_iterator::operator++() {
    i++;
    j++;
    return *this;
}

//---------------------------------------------------------------------------
matrix_diagonal_iterator matrix_diagonal_iterator::operator++(int) {
    i++;
```

```
    j++;
    return *this;
}

//-----------------------------------------------------------------------------
bool matrix_diagonal_iterator::operator==(const matrix_diagonal_iterator& b) const {
    return b.i == i && b.j == j;
}

//-----------------------------------------------------------------------------
bool matrix_diagonal_iterator::operator!=(const matrix_diagonal_iterator& b) const {
    return b.i != i || b.j != j;
}

//-----------------------------------------------------------------------------
matrix_diagonal_iterator& matrix_diagonal_iterator::operator+=(const ptrdiff_t& movement) {
    i += movement;
    j += movement;
    return *this;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
matrix_diagonal_line_iterator::matrix_diagonal_line_iterator(int n, std::vector<int> format, bool
↪  isOnlyLowTriangle) : dc(n), m_isEnd(false), m_isLineEnd(false) {
    // Создаем обратное преобразование из формата диагонали в ее номер в формате
    for (int i = 0; i < format.size(); ++i)
        if ((isOnlyLowTriangle && format[i] < 0) || !isOnlyLowTriangle)
            m_map[format[i]] = i;

    // Создаем сортированный формат, чтобы по нему двигаться
    if (isOnlyLowTriangle) {
        for (int i = 0; i < format.size(); ++i)
            if (format[i] < 0)
                m_sorted_format.push_back(format[i]);
    } else
        m_sorted_format = format;
    std::sort(m_sorted_format.begin(), m_sorted_format.end());

    line = 0;
    pos = 0;
    start = m_sorted_format.size() - 1;
    end = start;

    // Находим, с какой диагонали начинается текущая строка
    for (int i = 0; i < m_sorted_format.size(); ++i) {
        if (dc.isLineIntersectDiagonal(line, m_sorted_format[i])) {
            start = i;
            break;
        }
    }

    // Находим на какой диагонали кончается текущая строка
    for (int i = 0; i < m_sorted_format.size(); ++i) {
        int j = m_sorted_format.size() - i - 1;
        if (dc.isLineIntersectDiagonal(line, m_sorted_format[j])) {
            end = j;
            break;
        }
    }

    calcPos();
}

//-----------------------------------------------------------------------------
matrix_diagonal_line_iterator& matrix_diagonal_line_iterator::operator++() {
    if (!m_isEnd) {
        if (m_isLineEnd) {
            // Сдвигаемся на одну строку
            line++;

            // Определяем какие диагонали пересекают эту строку
            if (start != 0)
                if (dc.isLineIntersectDiagonal(line, m_sorted_format[start-1]))
                    start = start-1;

            if (end != 0)
                if (!dc.isLineIntersectDiagonal(line, m_sorted_format[end]))
                    if (start != end)
                        end = end-1;

            m_isLineEnd = false;
            if (line == dc.n)
                m_isEnd = true;

            pos = 0;
```

```
            calcPos();
        } else {
            // Сдвигаемся на один столбец
            pos++;
            calcPos();
        }
    }

    return *this;
}

//---------------------------------------------------------------------------
matrix_diagonal_line_iterator matrix_diagonal_line_iterator::operator++(int) {
    return operator++();
}

//---------------------------------------------------------------------------
bool matrix_diagonal_line_iterator::isLineEnd(void) const {
    return m_isLineEnd;
}

//---------------------------------------------------------------------------
bool matrix_diagonal_line_iterator::isEnd(void) const {
    return m_isEnd;
}

//---------------------------------------------------------------------------
void matrix_diagonal_line_iterator::calcPos(void) {
    // Вычисляет все текущие положения согласно переменным start, pos и формату
    if (!dc.isLineIntersectDiagonal(line, m_sorted_format[end]) || (start + pos > end)) {
        m_isLineEnd = true;
        i = line;
        j = 0;
        d = 0;
        di = 0;
        dn = 0;
    } else {
        i = line;
        d = m_sorted_format[start + pos];
        dn = m_map[d];
        di = dc.calcPos_byDL(d, i);
        j = dc.calcRow_byDL(d, i);
    }
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
std::vector<int> makeSevenDiagonalFormat(int n, int m, int k) {
    std::vector<int> result;

    if (1+m+k >= n)
        throw std::exception();

    result.push_back(0);

    result.push_back(1);
    result.push_back(1+m);
    result.push_back(1+m+k);

    result.push_back(-1);
    result.push_back(-1-m);
    result.push_back(-1-m-k);

    return result;
}

//---------------------------------------------------------------------------
std::vector<int> generateRandomFormat(int n, int diagonalsCount) {
    Diagonal d(n);

    std::vector<int> result;
    result.push_back(0);

    // Создаем массив всех возможных диагоналей
    std::vector<int> diagonals;
    for (int i = d.calcMinDiagonal(); i <= d.calcMaxDiagonal(); ++i)
        if (i != 0)
            diagonals.push_back(i);

    diagonalsCount = std::min<int>(diagonals.size(), diagonalsCount);

    // Заполняем результат случайными диагоналями из этого массива
    for (int i = 0; i < diagonalsCount; ++i) {
        int pos = intRandom(0, diagonals.size());
        result.push_back(diagonals[pos]);
        diagonals.erase(diagonals.begin() + pos);
```

```
    }

    return result;
}

//-------------------------------------------------------------------------------
void generateDiagonalMatrix(int n, int min, int max, std::vector<int> format, MatrixDiagonal& result) {
    result.resize(n, format);
    for (int i = 0; i < result.getDiagonalsCount(); ++i) {
        auto mit = result.posBegin(i);
        for (auto it = result.begin(i); it != result.end(i); ++it, ++mit)
            (*it) = intRandom(min, max);
    }
}

//-------------------------------------------------------------------------------
void generateDiagonallyDominantMatrix(int n, std::vector<int> format, bool isNegative,
↪  MatrixDiagonal& result) {
    result.resize(n, format);

    for (int i = 0; i < result.getDiagonalsCount(); ++i) {
        auto mit = result.posBegin(i);
        for (auto it = result.begin(i); it != result.end(i); ++it, ++mit) {
            if (isNegative)
                *it = -intRandom(0, 5);
            else
                *it = intRandom(0, 5);
        }
    }

    matrix_diagonal_line_iterator mit(n, format, false);
    for (; !mit.isEnd(); ++mit) {
        sumreal& sum = result.begin(0)[mit.i];
        sum = 0;
        for (; !mit.isLineEnd(); ++mit)
            if (mit.i != mit.j)
                sum += result.begin(mit.dn)[mit.di];
        sum = std::fabs(sum);
    }

    result.begin(0)[0] += 1;
}

//-------------------------------------------------------------------------------
bool mul(const MatrixDiagonal& a, const Vector& x, Vector& y) {
    if (x.size() != a.dimension())
        return false;

    y.resize(x.size());

    // Зануление результата
    y.zero();
    for (int i = 0; i < a.getDiagonalsCount(); ++i) {
        auto mit = a.posBegin(i);
        for (auto it = a.begin(i); it != a.end(i); ++it, ++mit)
            y(mit.i) += (*it) * x(mit.j);
    }

    return true;
}

//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
SolverSLAE_Iterative::SolverSLAE_Iterative() :
    w(1),
    isLog(false),
    log(std::cout),
    start(),
    epsilon(0.00001),
    maxIterations(100) {
}

//-------------------------------------------------------------------------------
void SolverSLAE_Iterative::save(std::ostream& out) const {
    out << w << std::endl;
    out << isLog << std::endl;
    start.save(out);
    out << std::scientific;
    out << epsilon << std::endl;
    out << std::defaultfloat;
    out << maxIterations << std::endl;
}

//-------------------------------------------------------------------------------
void SolverSLAE_Iterative::load(std::istream& in) {
```

```
    in >> w >> isLog;
    start.load(in);
    in >> epsilon >> maxIterations;
}

//-------------------------------------------------------------------------------
IterationsResult SolverSLAE_Iterative::jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x)
↪  const {
    return iteration_process(a, y, x, &SolverSLAE_Iterative::iteration_jacobi);
}

//-------------------------------------------------------------------------------
IterationsResult SolverSLAE_Iterative::seidel(const MatrixDiagonal& a, const Vector& y, Vector& x)
↪  const {
    return iteration_process(a, y, x, &SolverSLAE_Iterative::iteration_seidel);
}

//-------------------------------------------------------------------------------
IterationsResult SolverSLAE_Iterative::iteration_process(const MatrixDiagonal& a, const Vector& y,
↪  Vector& x, step_function step) const {
    if (a.dimension() != y.size() || start.size() != y.size())
        throw std::exception();

    // Считаем норму матрицы: ее максимальный элемент по модулю
    real yNorm = calcNorm(y);
    x1.resize(y.size());
    x = start;

    // Цикл по итерациям
    int i = 0;
    real relativeResidual = epsilon + 1;
    for (; i < maxIterations && relativeResidual > epsilon; ++i) {
        // Итерационный шаг
        step(this, a, y, x);

        // Считаем невязку
        mul(a, x, x1);
        x1.negate();
        sum(x1, y, x1);
        relativeResidual = fabs(calcNorm(x1)) / yNorm;

        // Выводим данные
        if (isLog)
            log << i << "\t" << std::scientific << std::setprecision(3) << relativeResidual <<
            ↪  std::endl;
    }

    return {i, relativeResidual};
}

//-------------------------------------------------------------------------------
void SolverSLAE_Iterative::iteration_jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x) const
{
    // Умножаем матрицу на решение
    mul(a, x, x1);

    // x^(k+1) = x^k + w/a(i, i) * x^(k+1)
    auto it = a.begin(0);
    for (int i = 0; i < x1.size(); ++i, ++it)
        x(i) += w / (*it) * (y(i)-x1(i));
}

//-------------------------------------------------------------------------------
void SolverSLAE_Iterative::iteration_seidel(const MatrixDiagonal& a, const Vector& y, Vector& x) const
{
    // Умножем верхний треугольник на решение
    x1.zero();
    for (int i = 0; i < a.getDiagonalsCount(); ++i)
        if (a.getDiagonalPos(i) >= 0) {
            auto mit = a.posBegin(i);
            for (auto it = a.begin(i); it != a.end(i); ++it, ++mit)
                x1(mit.i) += (*it) * x(mit.j);
        }

    // Проходим по нижнему треугольнику и считаем все параметры
    matrix_diagonal_line_iterator mit(a.dimension(), a.getFormat(), true);
    for (; !mit.isEnd(); ++mit) {
        for (; !mit.isLineEnd(); ++mit)
            x1(mit.i) += a.begin(mit.dn)[mit.di] * x(mit.j);
        x(mit.i) = x(mit.i) + w/a.begin(0)[mit.i] * (y(mit.i) - x1(mit.i));
    }
}
```

FILE **vector.cpp**

```cpp
#include <fstream>
#include <iomanip>
#include <cmath>
#include "vector.h"

//-----------------------------------------------------------------------------
Vector::Vector() {
}

//-----------------------------------------------------------------------------
Vector::Vector(int size, real fill) : mas(size, fill) {
}

//-----------------------------------------------------------------------------
Vector::Vector(const Matrix& a) {
    if (a.width() == 1) {
        resize(a.height());
        for (int i = 0; i < a.height(); ++i)
            mas[i] = a(i, 0);
    } else if (a.height() == 1) {
        resize(a.width());
        for (int i = 0; i < a.width(); ++i)
            mas[i] = a(0, i);
    } else
        throw std::exception();
}

//-----------------------------------------------------------------------------
void Vector::toDenseMatrix(Matrix& dense, bool isVertical) const {
    if (isVertical) {
        dense.resize(1, mas.size());
        for (int i = 0; i < mas.size(); ++i)
            dense(i, 0) = mas[i];
    } else {
        dense.resize(mas.size(), 1);
        for (int i = 0; i < mas.size(); ++i)
            dense(0, i) = mas[i];
    }
}

//-----------------------------------------------------------------------------
void Vector::loadFromFile(std::string fileName) {
    std::ifstream fin(fileName);
    load(fin);
    fin.close();
}

//-----------------------------------------------------------------------------
void Vector::saveToFile(std::string fileName) const {
    std::ofstream fout(fileName);
    fout.precision(std::numeric_limits<real>::digits10);
    save(fout);
    fout.close();
}

//-----------------------------------------------------------------------------
void Vector::save(std::ostream& out) const {
    out << mas.size() << std::endl;
    for (const auto& i : mas)
        out << i << std::endl;
    out << std::endl;
}

//-----------------------------------------------------------------------------
void Vector::load(std::istream& in) {
    int n;
    in >> n;
    resize(n);
    for (int i = 0; i < n; ++i)
        in >> mas[i];
}

//-----------------------------------------------------------------------------
void Vector::resize(int n, real fill) {
    if (mas.size() != n)
        mas.resize(n, fill);
}

//-----------------------------------------------------------------------------
void Vector::negate(void) {
    for (auto& i : mas)
        i = -i;
}

//-----------------------------------------------------------------------------
void Vector::zero(void) {
    for (auto& i : mas)
```

```
        i = 0;
}

//---------------------------------------------------------------------------
void Vector::generate(int n, int min, int max) {
    resize(n);
    for (int i = 0; i < n; ++i)
        operator()(i) = intRandom(min, max);
}

//---------------------------------------------------------------------------
void Vector::generate(int n) {
    resize(n);
    for (int i = 0; i < n; ++i)
        operator()(i) = i+1;
}

//---------------------------------------------------------------------------
int Vector::size(void) const {
    return mas.size();
}

//---------------------------------------------------------------------------
real& Vector::operator()(int i) {
    return mas[i];
}

//---------------------------------------------------------------------------
const real& Vector::operator()(int i) const {
    return mas[i];
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
bool sum(const Vector& a, const Vector& b, Vector& result) {
    if (a.size() != b.size())
        return false;

    if (result.size() != a.size())
        result.resize(a.size());

    for (int i = 0; i < result.size(); ++i)
        result(i) = a(i) + b(i);

    return true;
}

//---------------------------------------------------------------------------
sumreal sumAllElementsAbs(const Vector& a) {
    sumreal sum = 0;
    for (int i = 0; i < a.size(); ++i)
        sum += fabs(a(i));

    return sum;
}

//---------------------------------------------------------------------------
bool mul(const Matrix& a, const Vector& b, Vector& result) {
    // result = a * b
    if (a.width() != b.size())
        return false;

    result.resize(b.size());

    for (int i = 0; i < a.height(); ++i) {
        sumreal sum = 0;
        for (int j = 0; j < a.width(); ++j)
            sum += a(i, j) * b(j);
        result(i) = sum;
    }

    return true;
}

//---------------------------------------------------------------------------
real calcNorm(const Vector& a) {
    sumreal sum = 0;
    for (int i = 0; i < a.size(); ++i)
        sum += a(i) * a(i);

    return std::sqrt(sum);
}
```

FILE **common.cpp**

```cpp
#include <cmath>
#include "common.h"

//---------------------------------------------------------------------------
bool isNear(double a, double b) {
    if (a != 0) {
        if (fabs(a - b)/a > 0.0001)
            return false;
    } else {
        if (fabs(b) > 0.0001)
            return false;
    }

    return true;
}

//---------------------------------------------------------------------------
double random(void) {
    return std::rand() / double(RAND_MAX);
}

//---------------------------------------------------------------------------
int intRandom(int min, int max) {
    return min + random() * (max - min);
}
```

FILE **matrix.cpp**

```cpp
#include <fstream>
#include <iomanip>
#include "matrix.h"

//---------------------------------------------------------------------------
Matrix::Matrix(int n, int m, real fill) : m_matrix(m, std::vector<real>(n, fill)), m_n(n), m_m(m) {
}

//---------------------------------------------------------------------------
void Matrix::loadFromFile(std::string fileName) {
    std::ifstream fin(fileName);

    m_matrix.clear();
    int n, m;
    fin >> n >> m;
    resize(n, m);
    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j < width(); ++j) {
            fin >> operator()(i, j);
        }
    }

    fin.close();
}

//---------------------------------------------------------------------------
void Matrix::saveToFile(std::string fileName) const {
    std::ofstream fout(fileName);

    fout.precision(std::numeric_limits<real>::digits10);
    int w = std::numeric_limits<real>::digits10 + 4;
    save(fout);

    fout.close();
}

//---------------------------------------------------------------------------
void Matrix::load(std::istream& in) {
    m_matrix.clear(); m_m = 0; m_n = 0;
    int n, m;
    in >> n >> m;
    resize(n, m);
    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j < width(); ++j) {
            in >> operator()(i, j);
        }
    }
}

//---------------------------------------------------------------------------
void Matrix::save(std::ostream& out) const {
    out << m_n << "\t" << m_m << std::endl;
    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j < width(); ++j)
            out << "\t" << std::setw(10) << operator()(i, j);
        out << std::endl;
```

```
    }
    out << std::endl;
}


//-----------------------------------------------------------------------------
void Matrix::getFromVector(int n, int m, const std::vector<real>& data) {
    resize(n, m);
    for (int i = 0; i < data.size(); ++i)
        operator()(i / n, i % n) = data[i];
}

//-----------------------------------------------------------------------------
void Matrix::resize(int n, int m, real fill) {
    if (m_n != n || m_m != m) {
        m_n = n;
        m_m = m;
        m_matrix.clear();
        m_matrix.resize(m_m, std::vector<real>(m_n, fill));
    }
}

//-----------------------------------------------------------------------------
void Matrix::negate(void) {
    for (auto& i : m_matrix) {
        for (auto& j : i) {
            j = -j;
        }
    }
}

//-----------------------------------------------------------------------------
bool Matrix::isSymmetric(void) const {
    if (height() != width())
        return false;

    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j <= i; ++j) {
            const real& a = operator()(i, j);
            const real& b = operator()(j, i);
            if (!isNear(a, b))
                return false;
        }
    }

    return true;
}

//-----------------------------------------------------------------------------
bool Matrix::isLowerTriangular(void) const {
    if (height() != width())
        return false;

    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j < i; ++j) {
            if (fabs(operator()(j, i)) > 0.000001)
                return false;
        }
    }

    return true;
}

//-----------------------------------------------------------------------------
bool Matrix::isUpperTriangular(void) const {
    if (height() != width())
        return false;

    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j < i; ++j) {
            if (operator()(i, j) != 0)
                return false;
        }
    }

    return true;
}

//-----------------------------------------------------------------------------
bool Matrix::isDiagonal(void) const {
    if (height() != width())
        return false;

    for (int i = 0; i < height(); ++i) {
        for (int j = 0; j < i; ++j) {
            if (operator()(j, i) != 0 && operator()(i, j) != 0)
                return false;
        }
    }
```

```
    return true;
}

//---------------------------------------------------------------------------
bool Matrix::isDiagonalIdentity(void) const {
    if (height() != width())
        return false;

    for (int i = 0; i < height(); ++i) {
        if (operator()(i, i) != 1)
            return false;
    }

    return true;
}

//---------------------------------------------------------------------------
bool Matrix::isDegenerate(void) const {
    // TODO
    return false;
}

//---------------------------------------------------------------------------
real& Matrix::operator()(int i, int j) {
    return m_matrix[i][j];
}

//---------------------------------------------------------------------------
const real& Matrix::operator()(int i, int j) const {
    return m_matrix[i][j];
}

//---------------------------------------------------------------------------
int Matrix::width(void) const {
    return m_n;
}

//---------------------------------------------------------------------------
int Matrix::height(void) const {
    return m_m;
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
void generateSparseSymmetricMatrix(int n, int min, int max, real percent, Matrix& result) {
    result.resize(n, n, 0);

    int count = percent * n * n;

    for (int k = 0; k < count; ++k) {
        int i = intRandom(0, n-1);
        int j = intRandom(0, i-1);
        result(i, j) = intRandom(min, max);
        result(j, i) = result(i, j);
    }

    for (int i = 0; i < n; i++)
        result(i, i) = intRandom(1, max - min);
}

//---------------------------------------------------------------------------
void generateLMatrix(int n, int min, int max, real percent, Matrix& result) {
    result.resize(n, n, 0);

    int count = percent * n * n / 2;

    for (int k = 0; k < count; ++k) {
        int i = intRandom(0, n-1);
        int j = intRandom(0, i-1);
        result(i, j) = intRandom(min, max);
    }

    for (int i = 0; i < n; ++i) {
        result(i, i) = 1;
    }
}

//---------------------------------------------------------------------------
void generateDiagonalMatrix(int n, int min, int max, Matrix& result) {
    result.resize(n, n, 0);

    for (int i = 0; i < n; ++i)
        result(i, i) = intRandom(min, max);
}
```

```
//--------------------------------------------------------------------------------
void generateVector(int n, int min, int max, Matrix& result) {
    result.resize(1, n, 0);

    for (int i = 0; i < n; ++i)
        result(i, 0) = intRandom(min, max);
}

//--------------------------------------------------------------------------------
void generateVector(int n, Matrix& result) {
    result.resize(1, n, 0);

    for (int i = 0; i < n; ++i)
        result(i, 0) = i+1;
}

//--------------------------------------------------------------------------------
void generateGilbertMatrix(int n, Matrix& result) {
    result.resize(n, n);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result(i, j) = double(1.0)/double((i+1)+(j+1)-1);
        }
    }
}

//--------------------------------------------------------------------------------
void generateTestMatrix(int n, int profileSize, Matrix& result) {
    result.resize(n, n);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < profileSize; ++j) if (i-j-1 >= 0) {
            result(i, i-j-1) = -intRandom(0, 5);
            result(i-j-1, i) = result(i, i-j-1);
        }
    }

    for (int i = 0; i < n; ++i) {
        sumreal sum = 0;
        for (int j = 0; j < n; ++j) if (i != j) {
            sum += result(i, j);
        }
        result(i, i) = -sum;
    }
}

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------

//--------------------------------------------------------------------------------
bool mul(const Matrix& a, const Matrix& b, Matrix& result) {
    // result = rus_a * b
    if (a.width() != b.height())
        return false;

    result.resize(b.width(), a.height());

    for (int i = 0; i < b.width(); ++i) {
        for (int j = 0; j < a.height(); ++j) {
            real sum = 0;
            for (int k = 0; k < a.width(); ++k) {
                sum += a(j, k) * b(k, i);
            }
            result(j, i) = sum;
        }
    }

    return true;
}

//--------------------------------------------------------------------------------
bool sum(const Matrix& a, const Matrix& b, Matrix& result) {
    // result = rus_a + b
    if (a.width() != b.width() || a.height() != b.height())
        return false;

    result.resize(a.width(), a.height());

    for (int i = 0; i < a.width(); ++i) {
        for (int j = 0; j < a.height(); ++j) {
            result(j, i) = a(j, i) + b(j, i);
        }
    }

    return true;
}

```

```
//---------------------------------------------------------------------------
bool transpose(Matrix& a) {
    // rus_a = rus_a^T
    Matrix a_t(a.height(), a.width());

    for (int i = 0; i < a.height(); ++i) {
        for (int j = 0; j < a.width(); ++j) {
            a_t(j, i) = a(i, j);
        }
    }

    a = a_t;

    return true;
}

//---------------------------------------------------------------------------
sumreal sumAllElementsAbs(const Matrix& a) {
    sumreal sum = 0;
    for (int i = 0; i < a.height(); ++i) {
        for (int j = 0; j < a.width(); ++j) {
            sum += fabs(a(i, j));
        }
    }

    return sum;
}

//---------------------------------------------------------------------------
bool calcLUsq(const Matrix& a, Matrix& l, Matrix& u) {
    if (a.width() != a.height())
        return false;

    l.resize(a.width(), a.height(), 0);
    u.resize(a.width(), a.height(), 0);

    for (int i = 0; i < a.height(); ++i) {
        // Считаем элементы матрицы L
        for (int j = 0; j < i; ++j) {
            real sum = 0;
            for (int k = 0; k < j; ++k)
                sum += l(i, k) * u(k, j);

            l(i, j) = (a(i, j) - sum) / l(j, j);
        }

        // Считаем элементы матрицы U
        for (int j = 0; j < i; ++j) {
            real sum = 0;
            for (int k = 0; k < j; ++k)
                sum += l(j, k) * u(k, i);

            u(j, i) = (a(j, i) - sum) / u(j, j);
        }

        // Считаем диагональный элемент
        real sum = 0;
        for (int k = 0; k < i; ++k)
            sum += l(i, k) * u(k, i);

        l(i, i) = sqrt(a(i, i) - sum);
        u(i, i) = l(i, i);
    }

    return true;
}

//---------------------------------------------------------------------------
bool calcLUsq_partial(const Matrix& a, Matrix& l, Matrix& u) {
    if (a.width() != a.height())
        return false;

    l.resize(a.width(), a.height(), 0);
    u.resize(a.width(), a.height(), 0);

    for (int i = 0; i < a.height(); ++i) {
        // Считаем элементы матрицы L
        for (int j = 0; j < i; ++j) if (a(i, j) != 0) {
            real sum = 0;
            for (int k = 0; k < j; ++k)
                sum += l(i, k) * u(k, j);

            l(i, j) = (a(i, j) - sum) / l(j, j);
        }

        // Считаем элементы матрицы U
        for (int j = 0; j < i; ++j) if (a(j, i) != 0) {
            real sum = 0;
            for (int k = 0; k < j; ++k)
```

```
                sum += l(j, k) * u(k, i);

            u(j, i) = (a(j, i) - sum) / u(j, j);
        }

        // Считаем диагональный элемент
        real sum = 0;
        for (int k = 0; k < i; ++k)
            sum += l(i, k) * u(k, i);

        l(i, i) = sqrt(a(i, i) - sum);
        u(i, i) = l(i, i);
    }

    return true;
}

//---------------------------------------------------------------------------
bool calcLDL(const Matrix& a, Matrix& l, Matrix& d) {
    // l * d * l^T = rus_a
    if (!a.isSymmetric())
        return false;

    l.resize(a.width(), a.height(), 0);
    d.resize(a.width(), a.height(), 0);

    for (int i = 0; i < a.height(); ++i) {
        // Считаем элементы матрицы L
        for (int j = 0; j < i; ++j) {
            real sum = 0;
            for (int k = 0; k < j; ++k)
                sum += d(k, k) * l(j, k) * l(i, k);

            if (fabs(d(j, j)) < 0.0001)
                l(i, j) = 0;
            else
                l(i, j) = (a(i, j) - sum) / d(j, j);
        }

        // Считаем диагональный элемент
        {
            real sum = 0;
            for (int j = 0; j < i; ++j)
                sum += d(j, j) * l(i, j) * l(i, j);
            d(i, i) = a(i, i) - sum;
        }
    }

    for (int i = 0; i < l.height(); i++)
        l(i, i) = 1;

    return true;
}

//---------------------------------------------------------------------------
bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x) {
    // l * x = y, l - нижнетреугольная матрица
    if (!l.isLowerTriangular() || !l.isDiagonalIdentity())
        return false;

    x.resize(1, y.height());

    for (int i = x.height() - 1; i >= 0; --i) {
        real sum = 0;
        for (int j = i; j < x.height(); ++j)
            sum += l(j, i) * x(j, 0);
        x(i, 0) = y(i, 0) - sum;
    }

    return true;
}

//---------------------------------------------------------------------------
bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x) {
    // l * x = y, l - верхнетреугольная матрица
    if (!l.isLowerTriangular() || !l.isDiagonalIdentity())
        return false;

    x.resize(1, y.height());

    for (int i = 0; i < x.height(); ++i) {
        real sum = 0;
        for (int j = 0; j < i; ++j)
            sum += l(i, j) * x(j, 0);
        x(i, 0) = y(i, 0) - sum;
    }

    return true;
}
```

```
//---------------------------------------------------------------------------
bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x) {
    // d * x = y, d - диагональная матрица
    if (!d.isDiagonal())
        return false;

    x.resize(1, y.height());

    for (int i = 0; i < x.height(); ++i)
        x(i, 0) = y(i, 0) / d(i, i);

    return true;
}

//---------------------------------------------------------------------------
bool solveSLAE_by_LDL(const Matrix& a, const Matrix& y, Matrix& x) {
    // rus_a * x = y, rus_a - симметричная матрица
    if (!(a.width() == a.height() && a.width() == y.height() && !a.isDegenerate()))
        return false;

    Matrix l, d, z, w;

    if (!calcLDL(a, l, d))
        return false;

    if (!calcGaussianFrontOrder(l, y, z))
        return false;

    if (!calcGaussianCentralOrder(d, z, w))
        return false;

    if (!calcGaussianReverseOrder(l, w, x))
        return false;

    return true;
}

//---------------------------------------------------------------------------
bool solveSLAE_byGaussMethod(const Matrix& a1, const Matrix& y1, Matrix& x1) {
    if (!(a1.width() == a1.height() && y1.height() == a1.width() && !a1.isDegenerate()))
        return false;

    Matrix a(a1);
    Matrix y(y1);

    for (int i = 0; i < a.height(); ++i) {
        // Находим максимальный элемент
        int maxI = i;
        for (int j = i+1; j < a.height(); ++j)
            if (fabs(a(j, i)) > fabs(a(maxI, i)))
                maxI = j;

        // Переставляем эту строчку с текущей
        for (int j = i; j < a.width(); ++j)
            std::swap(a(i, j), a(maxI, j));
        std::swap(y(i, 0), y(maxI, 0));

        // Перебираем все строчки ниже и отнимаем текущую строчку от них
        for (int j = i+1; j < a.height(); ++j) {
            real m = a(j, i) / a(i, i);
            for (int k = i; k < a.width(); ++k)
                a(j, k) -= m * a(i, k);
            y(j, 0) -= m * y(i, 0);
        }

        // Делим текущую строку на ее ведущий элемент, чтобы на диагонали были единицы
        double m = a(i, i);
        for (int j = i; j < a.width(); ++j)
            a(i, j) /= m;
        y(i, 0) /= m;
    }

    // Считаем обратный ход Гаусса
    transpose(a);
    calcGaussianReverseOrder(a, y, x1);

    return true;
}
```