

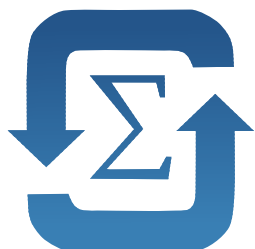
Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



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 реализованных 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.

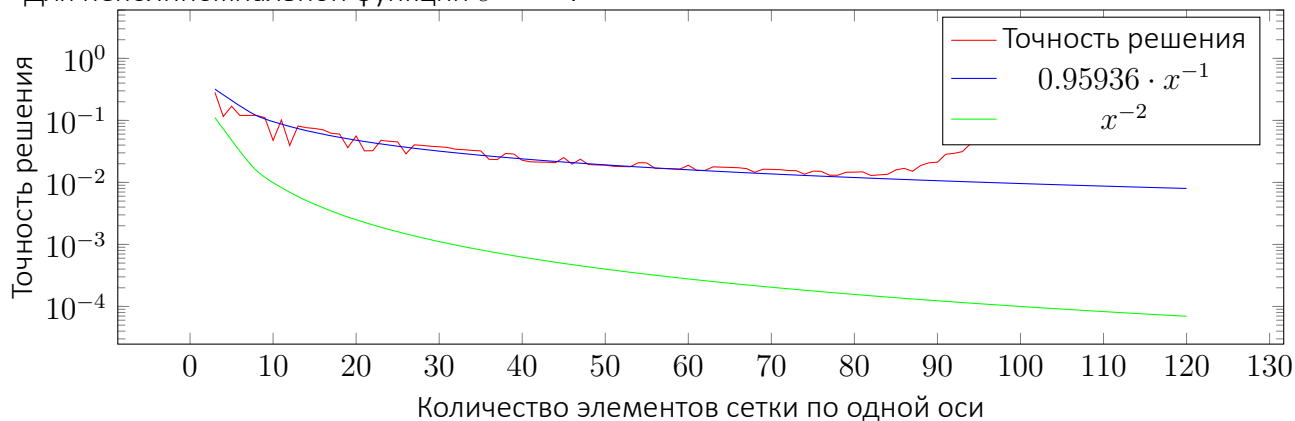
4.1 Точность для разных функций

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
$\sin x + \cos y$	$2.4 \cdot 10^{-4}$	$1.77 \cdot 10^{-4}$	0.31
$\sqrt{x^2 + y^2}$	$4.75 \cdot 10^{-3}$	$4.59 \cdot 10^{-3}$	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 $e^{x^3+yx^2}$.

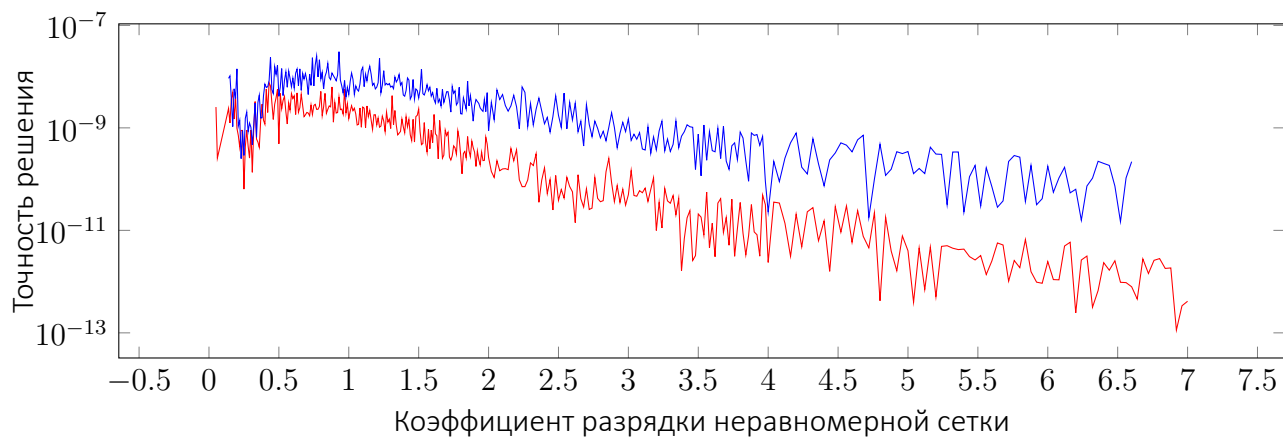


4.3 Зависимость точности от параметра разрядки неравномерной сетки

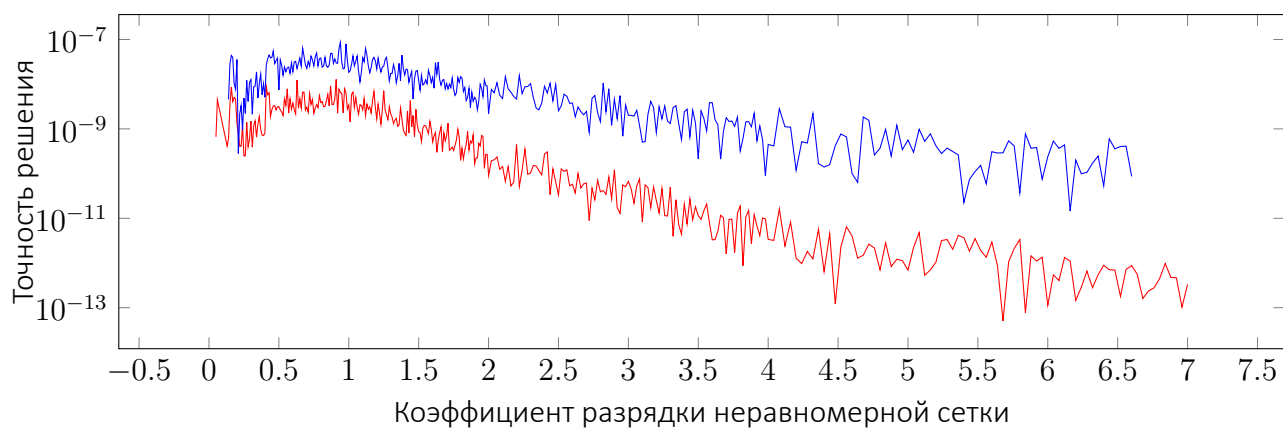
Параметры остаются прежними, с небольшими изменениями:

- 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.

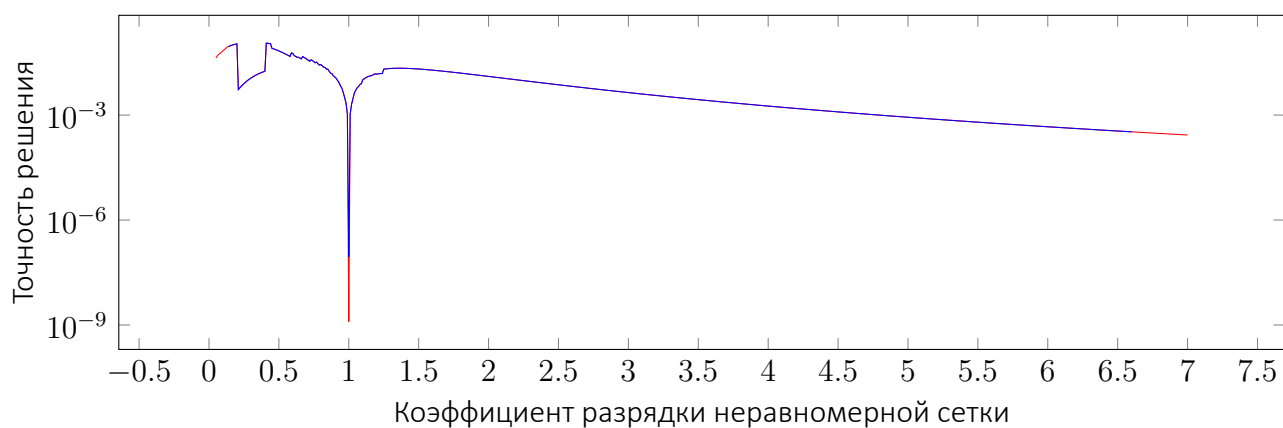
4.3.1 $2x + y$



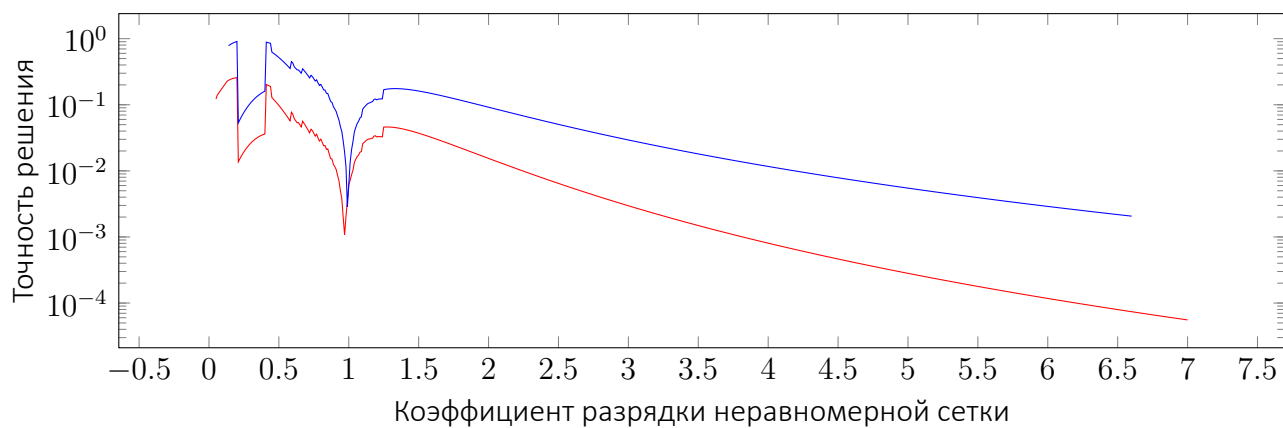
4.3.2 $3x^2 + y^2$



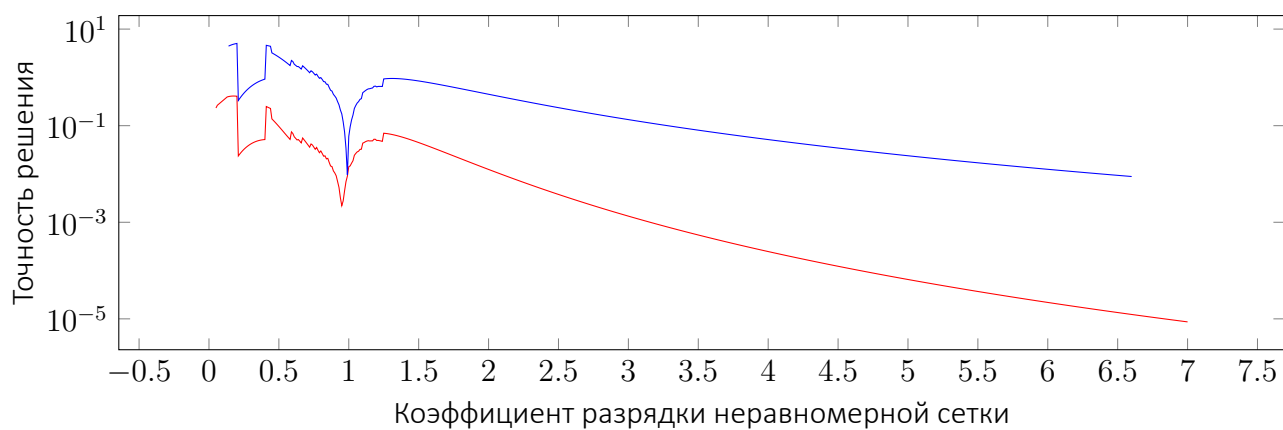
4.3.3 $x^3 + xy^2 + y^3$



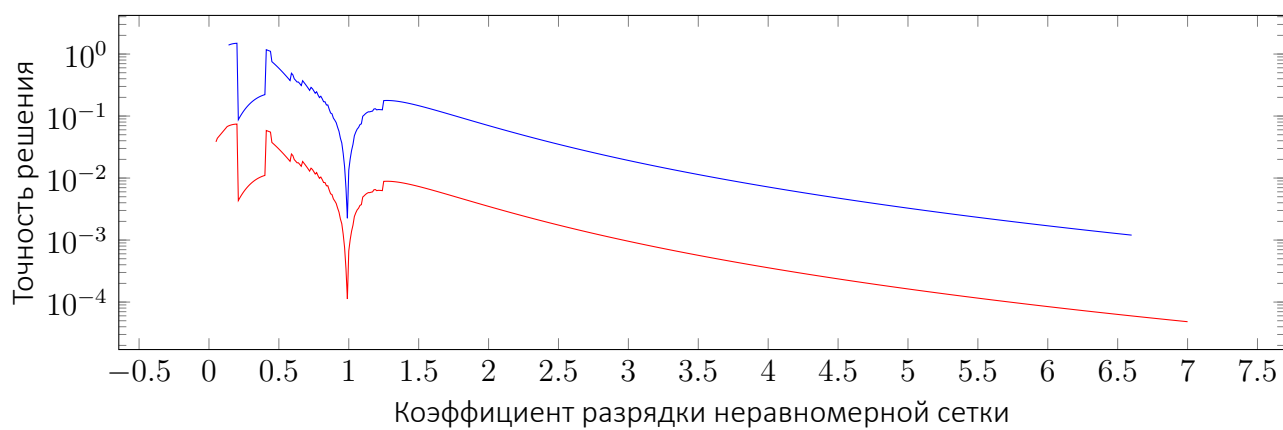
4.3.4 $x^4 + y^4$



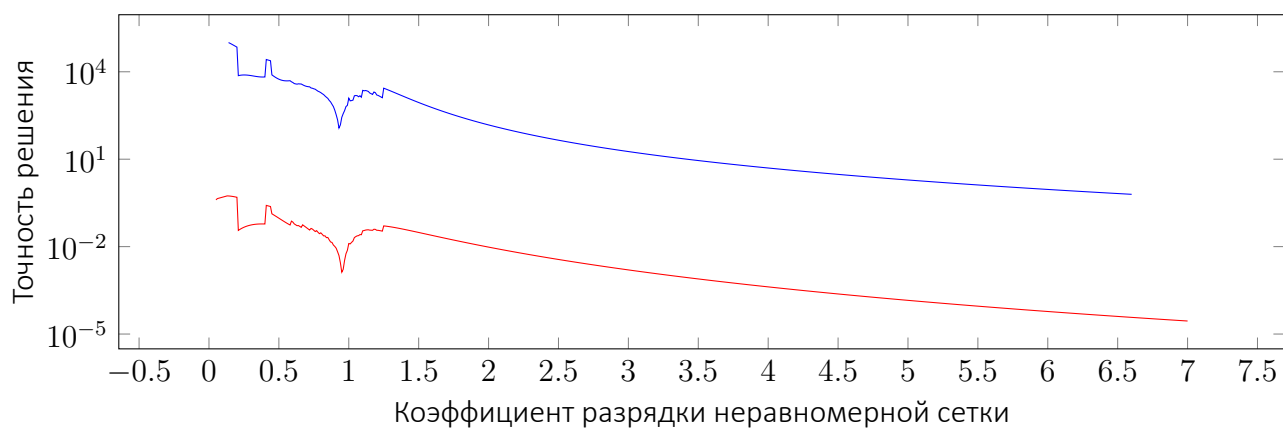
4.3.5 $x^5 + y^5 + 2xy$



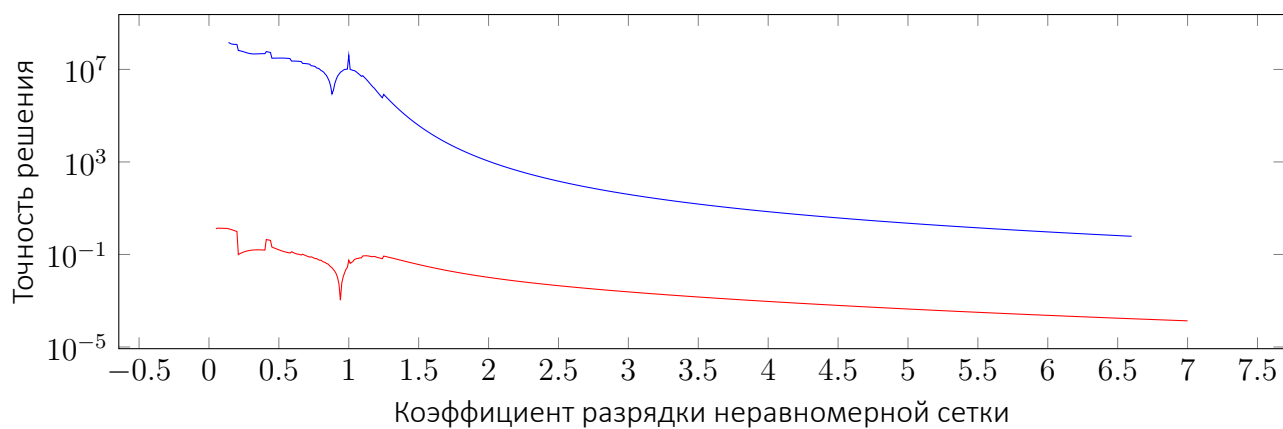
4.3.6 e^{x+y}



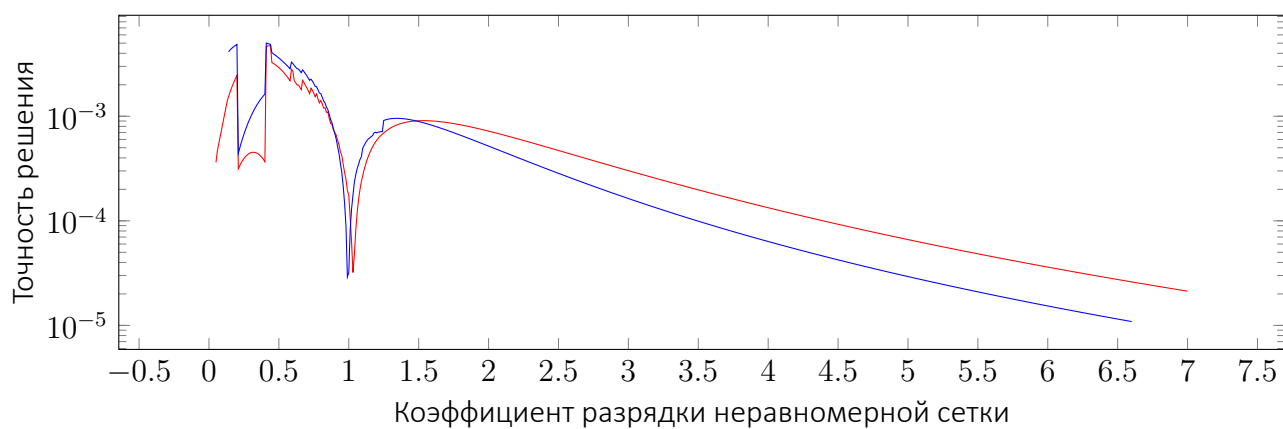
4.3.7 $e^{x^2+y^2}$



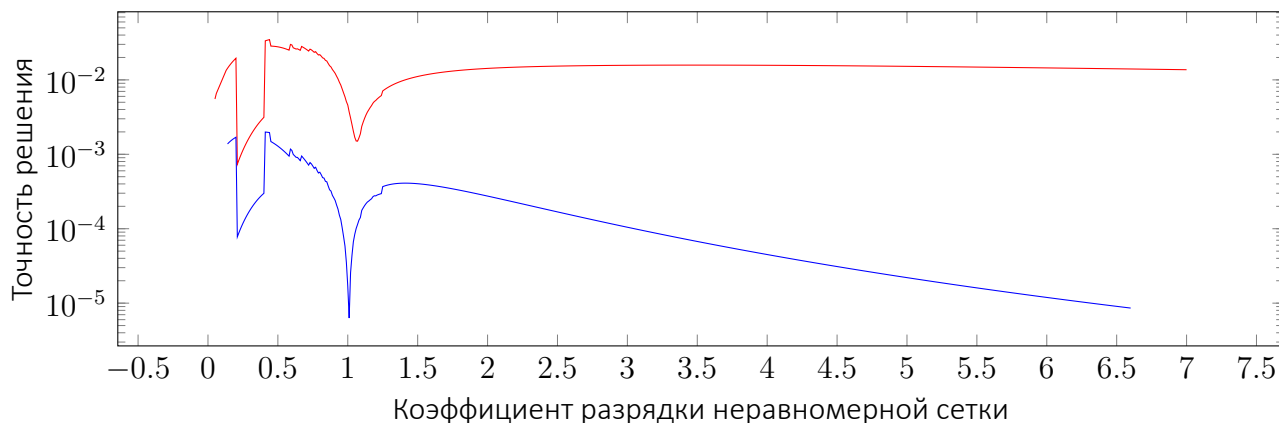
4.3.8 $e^{x^3+yx^2}$



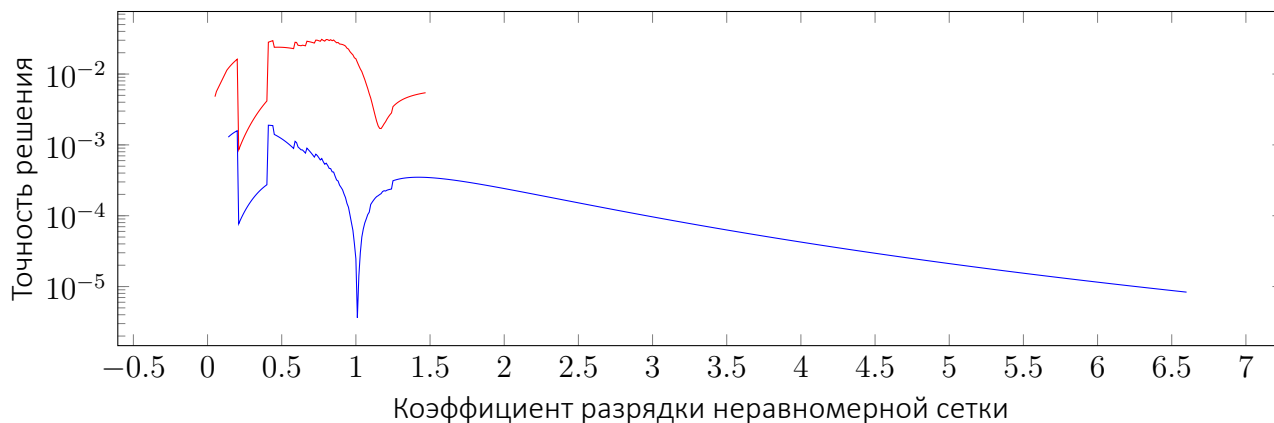
4.3.9 $\sin x + \cos y$



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

```
1 #include <iostream>
2 #include <functional>
3 #include <fstream>
4 #include <algorithm>
5 #include <cassert>
6 #include <diagonal.h>
7
8 typedef std::function<double(double)> Function1D;
9 typedef std::function<double(double, double)> Function2D;
10 typedef std::pair<Function2D, std::string> NamedFunction;
11
12 struct Cell
13 {
14     int i;
15
16     enum
17     {
18         FICTITIOUS, // Фиктивный узел
19         INNER, // Нет краевых условий
20         FIRST, //  $f(x, y) = \text{value}$ 
21         SECOND, //  $f'(x, y) = \text{value}$ 
22         THIRD, //  $f'(x, y) + c1 * f(x, y) + c2 = 0$ 
23     }
24     conditionType; // краевое условие
25
26     union {
27         struct {
28             double value;
29         } first;
30         struct {
31             Cell* cell_around;
32             double value;
33             double h;
34         } second;
35         struct {
36             Cell* cell_around;
37             double value;
38             double h;
39             double c1;
40         } third;
41     } conditionValue;
42
43     double value;
44     double x, y;
45
46     Cell *up, *down, *left, *right;
47 };
48
49 typedef std::vector<Cell> Cells;
50
51 class Field
52 {
53 public:
54     Field(double startx, double starty, double sizex, double sizey) : sizex(sizex), sizey(sizey),
55         ↪ startx(startx), starty(starty) {}
56
57     virtual bool isPointInside(double x, double y) const = 0;
58
59     double sizex, sizey;
60     double startx, starty;
61 };
62
63 bool isInsideSquare(double x, double y, double sizex, double sizey, double startx, double starty) {
64     x -= startx;
65     y -= starty;
66     return x >= 0 && x <= sizex && y >= 0 && y <= sizey;
67 }
68
69 class SquareField : public Field
70 {
71 public:
72     SquareField(double startx, double starty, double sizex, double sizey) : Field(startx, starty,
73         ↪ sizex, sizey) {}
74
75     bool isPointInside(double x, double y) const {
76         if (isInsideSquare(x, y, sizex, sizey, startx, starty))
77             return true;
78         else
79             return false;
80     }
81 };
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```



```

81 class ShField : public Field
82 {
83 public:
84     ShField(double startx, double starty, double sizex, double sizey) : Field(startx, starty, sizex,
85         ↪ sizey) {
86     }
87     bool isPointInside(double x, double y) const {
88         if (isInsideSquare(x, y, sizex, sizey, startx, starty) &&
89             ↪ !isInsideSquare(x, y, sizex * 0.2, sizey * 0.2, startx + sizex * 0.2, starty + sizey *
90             ↪ 0.8) &&
91             !isInsideSquare(x, y, sizex * 0.2, sizey * 0.2, startx + sizex * 0.6, starty + sizey * 0.8)
92         ))
93             return true;
94         else
95             return false;
96     }
97 };
98
99 class Grid
100 {
101 public:
102     virtual Cells makeCells(const Field& field,
103         ↪ double startx, double starty,
104         ↪ double sizex, double sizey,
105         ↪ int isizex, int isizey) = 0;
106 };
107
108 class NonUniformGrid : public Grid
109 {
110 public:
111     NonUniformGrid(double c) : c(c) {}
112
113     Cells makeCells(const Field& field,
114         ↪ double startx, double starty,
115         ↪ double sizex, double sizey,
116         ↪ int isizex, int isizey) {
117         Cells cells;
118         cells.reserve(isizex * isizey);
119         // [строка][столбец]
120         std::vector<std::vector<int>> grid(isizey, std::vector<int>(isizex, -1));
121
122         double x = startx, y = starty;
123         double hx, hy;
124         if (std::fabs(c - 1) > 0.00001) {
125             hx = sizex * (1.0 - c) / (1 - pow(c, isizex));
126             hy = sizey * (1.0 - c) / (1 - pow(c, isizey));
127         } else {
128             // Это равномерная сетка
129             hx = sizex / (isizex - 1);
130             hy = sizey / (isizey - 1);
131         }
132         double oldhx = hx;
133         int k = 0;
134         for (int i = 0; i < isizey; ++i, y += hy, hy *= c) {
135             x = startx;
136             hx = oldhx;
137             for (int j = 0; j < isizex; ++j, x += hx, hx *= c) {
138                 if (field.isPointInside(x, y)) {
139                     grid[i][j] = k;
140                     cells.push_back({k, Cell::INNER, 0, 0, x, y,
141                         ↪ nullptr,
142                         ↪ (i != 0 && grid[i-1][j] != -1) ? &cells[grid[i-1][j]] : nullptr,
143                         ↪ (j != 0 && grid[i][j-1] != -1) ? &cells[grid[i][j-1]] : nullptr,
144                         ↪ nullptr,
145                         ↪ });
146                     k++;
147                     if (i != 0 && grid[i-1][j] != -1) cells[grid[i-1][j]].up = &cells.back();
148                     if (j != 0 && grid[i][j-1] != -1) cells[grid[i][j-1]].right = &cells.back();
149                 } else {
150                     cells.push_back({k, Cell::FICTITIOUS, 0, 0, x, y, nullptr, nullptr, nullptr,
151                         ↪ nullptr});
152                     k++;
153                 }
154             }
155         }
156         return cells;
157     }
158 private:
159     double c;
160 };
161
162 class UniformGrid : public Grid
163 {
164 public:
165     UniformGrid() : grid(1.0) {}
166 }

```

```

165     Cells makeCells(const Field& field,
166                     double startx, double starty,
167                     double sizex, double sizey,
168                     int isizex, int isizey) {
169         return grid.makeCells(field, startx, starty, sizex, sizey, isizex, isizey);
170     }
171 private:
172     NonUniformGrid grid;
173 };
174
175 Function2D calcFirstDerivativeX(const Function2D& f) {
176     return [f] (double x, double y) -> double {
177         const double h = 0.00001;
178         return (-f(x+2*h, y)+8*f(x+h, y)-8*f(x-h, y)+f(x-2*h, y))/(12*h);
179     };
180 }
181
182 Function2D calcFirstDerivativeY(const Function2D& f) {
183     return [f] (double x, double y) -> double {
184         const double h = 0.00001;
185         return (-f(x, y+2*h)+8*f(x, y+h)-8*f(x, y-h)+f(x, y-2*h))/(12*h);
186     };
187 }
188
189 Function2D calcSecondDerivativeX(const Function2D& f) {
190     return [f] (double x, double y) -> double {
191         const double h = 0.0001;
192         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);
193     };
194 }
195
196 Function2D calcSecondDerivativeY(const Function2D& f) {
197     return [f] (double x, double y) -> double {
198         const double h = 0.0001;
199         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);
200     };
201 }
202
203 Function2D calcLaplacian(const Function2D& f) {
204     auto fxx = calcSecondDerivativeX(f);
205     auto fyy = calcSecondDerivativeY(f);
206     return [fxx, fyy] (double x, double y) -> double {
207         return fxx(x, y) + fyy(x, y);
208     };
209 }
210
211 void fillWithFunction(Cells& cells, const Function2D& f) {
212     for (auto& i : cells) {
213         i.value = f(i.x, i.y);
214     }
215 }
216
217 void fillBoundaryConditions1(Cells& cells, const Function2D& f) {
218     // Первые краевые условия
219     for (auto& i : cells) {
220         if (i.conditionType != Cell::FICTITIOUS && (i.up == nullptr || i.down == nullptr || i.left ==
221             ↪ nullptr || i.right == nullptr)) {
222             i.conditionType = Cell::FIRST;
223             i.conditionValue.first.value = f(i.x, i.y);
224         }
225     }
226 }
227
228 void fillBoundaryConditions2(Cells& cells, const Function2D& f) {
229     auto fx = calcFirstDerivativeX(f);
230     auto fy = calcFirstDerivativeY(f);
231     int isFirst = 2;
232     for (auto& i : cells) {
233         if (i.conditionType != Cell::FICTITIOUS && (i.up == nullptr || i.down == nullptr || i.left ==
234             ↪ nullptr || i.right == nullptr)) {
235             if (isFirst || i.left == nullptr && i.right == nullptr || i.up == nullptr && i.down ==
236                 ↪ nullptr) {
237                 // Тут уже ничего не сделаешь, потому что это палка, у которой нет соседей, чтобы
238                 ↪ можно было сделать краевые условия с производной
239                 i.conditionType = Cell::FIRST;
240                 i.conditionValue.first.value = f(i.x, i.y);
241                 isFirst--;
242             } else {
243                 i.conditionType = Cell::SECOND;
244                 double fxv = fx(i.x, i.y);
245                 double fyv = fy(i.x, i.y);
246
247                 Cell* cell_around;
248                 double& result = i.conditionValue.second.value;
249                 if (i.left == nullptr) {
250                     result = -fxv;
251                     cell_around = i.right;
252                 }

```

```

249     } else if (i.right == nullptr) {
250         result = fxv;
251         cell_around = i.left;
252     } else if (i.up == nullptr) {
253         result = fyv;
254         cell_around = i.down;
255     } else if (i.down == nullptr) {
256         result = -fyv;
257         cell_around = i.up;
258     } else {
259         throw std::exception("This is inner cell!");
260     }
261     i.conditionValue.second.cell_around = cell_around;
262
263     double hx = std::fabs(cell_around->x - i.x);
264     double hy = std::fabs(cell_around->y - i.y);
265     double h = std::max(hx, hy);
266     i.conditionValue.second.h = h;
267 }
268 }
269 }
270 }
271
272 void fillBoundaryConditions3(Cells& cells, const Function2D& f, double c1) {
273     auto fx = calcFirstDerivativeX(f);
274     auto fy = calcFirstDerivativeY(f);
275     for (auto& i : cells) {
276         if (i.conditionType != Cell::FICTITIOUS && (i.up == nullptr || i.down == nullptr || i.left ==
277             ↪ nullptr || i.right == nullptr)) {
278             if (i.left == nullptr && i.right == nullptr || i.up == nullptr && i.down == nullptr) {
279                 // Тут уже ничего не сделаешь, потому что это палка, у которой нет соседей, чтобы
280                 ↪ можно было сделать краевые условия с производной
281                 i.conditionType = Cell::FIRST;
282                 i.conditionValue.first.value = f(i.x, i.y);
283             } else {
284                 i.conditionType = Cell::THIRD;
285                 double fxv = fx(i.x, i.y);
286                 double fyv = fy(i.x, i.y);
287
288                 Cell* cell_around;
289                 double& result = i.conditionValue.third.value;
290                 if (i.left == nullptr) {
291                     result = -fxv;
292                     cell_around = i.right;
293                 } else if (i.right == nullptr) {
294                     result = fxv;
295                     cell_around = i.left;
296                 } else if (i.up == nullptr) {
297                     result = fyv;
298                     cell_around = i.down;
299                 } else if (i.down == nullptr) {
300                     result = -fyv;
301                     cell_around = i.up;
302                 } else {
303                     throw std::exception("This is inner cell!");
304                 }
305                 result += f(i.x, i.y)*c1;
306                 i.conditionValue.third.cell_around = cell_around;
307
308                 double hx = std::fabs(cell_around->x - i.x);
309                 double hy = std::fabs(cell_around->y - i.y);
310                 double h = std::max(hx, hy);
311                 i.conditionValue.third.h = h;
312                 i.conditionValue.third.c1 = c1;
313             }
314         }
315     }
316 }
317
318 double calcDifference(const Cells& a, const Cells& b) {
319     if (a.size() != b.size())
320         return -1;
321     double sum = 0;
322     for (int i = 0; i < a.size(); ++i) {
323         if (a[i].conditionType != Cell::FICTITIOUS) {
324             if (a[i].x != b[i].x || a[i].y != b[i].y)
325                 return -1;
326             sum += (a[i].value - b[i].value)*(a[i].value - b[i].value);
327         }
328     }
329     return std::sqrt(sum);
330 }
331
332 std::pair<MatrixDiagonal, Vector> makeSLAE(const Cells& cells, const Function2D& rightPart) {
333     int n = sqrt(cells.size());
334     assert(n*n == cells.size());
335     MatrixDiagonal A(cells.size(), { 0, 1, -1, n, -n });
336     Vector b(cells.size(), 0);

```

```

335 struct PushType
336 {
337     int i;
338     double value;
339 };
340
341 matrix_diagonal_line_iterator it(A.dimension(), A.getFormat(), false);
342 auto pushLine = [&](const std::vector<PushType>& p) {
343     for (; !it.isLineEnd(); ++it) {
344         for (auto& i : p) {
345             if (i.i == it.j)
346                 A.begin(it.dn)[it.di] = i.value;
347         }
348     }
349     ++it;
350 };
351
352 for (int i = 0; i < cells.size(); ++i) {
353     const auto& cell = cells[i];
354     switch (cell.conditionType) {
355         case Cell::FICTITIOUS: {
356             pushLine({{cell.i, 1}});
357             b(i) = 0;
358         } break;
359         case Cell::INNER: {
360             double uph = std::fabs(cell.up->y - cell.y);
361             double downh = std::fabs(cell.down->y - cell.y);
362             double lefth = std::fabs(cell.left->x - cell.x);
363             double righth = std::fabs(cell.right->x - cell.x);
364             pushLine({
365                 {cell.left->i, 2.0/(lefth*(righth + lefth))},
366                 {cell.down->i, 2.0/(downh*(uph + downh))},
367                 {cell.right->i, 2.0/(righth*(righth + lefth))},
368                 {cell.up->i, 2.0/(uph*(uph + downh))},
369                 {cell.i, -2.0/(uph*downh)-2.0/(lefth*righth)},
370             });
371             b(i) = rightPart(cell.x, cell.y);
372         } break;
373         case Cell::FIRST: {
374             pushLine({{cell.i, 1}});
375             b(i) = cell.conditionValue.first.value;
376         } break;
377         case Cell::SECOND: {
378             double h = cell.conditionValue.second.h;
379             pushLine({
380                 {cell.i, 1.0/h},
381                 {cell.conditionValue.second.cell_around->i, -1.0/h},
382             });
383             b(i) = cell.conditionValue.second.value;
384         } break;
385         case Cell::THIRD: {
386             double h = cell.conditionValue.third.h;
387             pushLine({
388                 {cell.i, 1.0/h + cell.conditionValue.third.c1},
389                 {cell.conditionValue.third.cell_around->i, -1.0/h},
390             });
391             b(i) = cell.conditionValue.third.value;
392         } break;
393     }
394 }
395
396 return {A, b};
397 }
398
399 Vector solveSLAE(std::pair<MatrixDiagonal, Vector>& slae) {
400     MatrixDiagonal& A = slae.first;
401     Vector& b = slae.second;
402     Vector result;
403     SolverSLAE_Iterative solver;
404     solver.epsilon = 1e-12;
405     solver.isLog = false;
406     solver.maxIterations = 3000;
407     solver.start = b;
408     solver.w = 1.4;
409     auto returned = solver.seidel(A, b, result);
410     //std::cout << returned.relativeResidual << std::endl;
411     return result;
412 }
413
414 void setCells(Cells& cells, const Vector& answer) {
415     for (int i = 0; i < cells.size(); i++)
416         cells[i].value = answer(i);
417 }
418
419 void make_table_nonuniform_grid(const std::string& name, const Field& field, const Function2D& f, int
420     size, int boundaryCondition) {

```

```

422 auto rightPart = calcLaplacian(f);
423 std::ofstream fout(name);
424 fout << "c\tnorm" << std::endl;
425 for (int i = 1; i <= 700; i++) {
426     NonUniformGrid grid(i/100.0);
427     auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizeX,
428     ↪ field.sizeY, size, size);
429     auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizeX,
430     ↪ field.sizeY, size, size);
431     fillWithFunction(cells_answer, f);
432     if (boundaryCondition == 1) {
433         fillBoundaryConditions1(cells_question, f);
434     } else if (boundaryCondition == 2) {
435         fillBoundaryConditions2(cells_question, f);
436     } else {
437         fillBoundaryConditions3(cells_question, f, 1);
438     }
439     auto slae = makeSLAE(cells_question, rightPart);
440     auto answer = solveSLAE(slae);
441     setCells(cells_question, answer);
442     double difference = calcDifference(cells_answer, cells_question);
443     if (difference == difference) {
444         fout << i / 100.0 << "\t" << difference << std::endl;
445     }
446     std::cout << "\r" << i;
447     if (i < 200) {
448         i++;
449     } else {
450         if (i < 400) {
451             i += 2;
452         } else {
453             if (i < 1000) {
454                 i += 4;
455             } else {
456                 i += 10;
457             }
458         }
459     }
460 }
461 fout.close();
462 }
463
464 void make_table_size(const std::string& name, const Field& field, const Function2D& f, int
465 ↪ boundaryCondition) {
466     auto rightPart = calcLaplacian(f);
467     std::ofstream fout(name);
468     fout << "size\tnorm" << std::endl;
469     for (int i = 3; i <= 120; i++) {
470         UniformGrid grid;
471         auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.sizeX,
472         ↪ field.sizeY, i, i);
473         auto cells_question = grid.makeCells(field, field.startx, field.starty, field.sizeX,
474         ↪ field.sizeY, i, i);
475         fillWithFunction(cells_answer, f);
476         if (boundaryCondition == 1) {
477             fillBoundaryConditions1(cells_question, f);
478         } else if (boundaryCondition == 2) {
479             fillBoundaryConditions2(cells_question, f);
480         } else {
481             fillBoundaryConditions3(cells_question, f, 1);
482         }
483         auto slae = makeSLAE(cells_question, rightPart);
484         auto answer = solveSLAE(slae);
485         setCells(cells_question, answer);
486         double difference = calcDifference(cells_answer, cells_question);
487         if (difference == difference) {
488             fout << i << "\t" << difference << std::endl;
489         }
490         std::cout << "\r" << i;
491     }
492     fout.close();
493 }
494
495 void make_table_conditions(const std::string& name, const Field& field, const
496 ↪ std::vector<NamedFunction>& namedFunctions) {
497     std::ofstream fout(name);
498     fout << "func\t1square\t1sh\t3sh" << std::endl;
499     SquareField field1(field.startx, field.starty, field.sizeX, field.sizeY);
500     ShField field2(field.startx, field.starty, field.sizeX, field.sizeY);
501     UniformGrid grid;
502     int size = 20;
503     for (auto& i : namedFunctions) {
504         fout << i.second << "\t";
505         auto& f = i.first;

```

```

504 auto rightPart = calcLaplacian(f);
505
506 {
507     auto& field = field1;
508     auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.size,
509     ↪ field.size, size, size);
510     auto cells_question = grid.makeCells(field, field.startx, field.starty, field.size,
511     ↪ field.size, size, size);
512     fillWithFunction(cells_answer, f);
513     fillBoundaryConditions1(cells_question, f);
514     auto slae = makeSLAE(cells_question, rightPart);
515     auto answer = solveSLAE(slae);
516     setCells(cells_question, answer);
517     double difference = calcDifference(cells_answer, cells_question);
518
519     fout << difference << "\t";
520 }
521
522 {
523     auto& field = field2;
524     auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.size,
525     ↪ field.size, size, size);
526     auto cells_question = grid.makeCells(field, field.startx, field.starty, field.size,
527     ↪ field.size, size, size);
528     fillWithFunction(cells_answer, f);
529     fillBoundaryConditions1(cells_question, f);
530     auto slae = makeSLAE(cells_question, rightPart);
531     auto answer = solveSLAE(slae);
532     setCells(cells_question, answer);
533     double difference = calcDifference(cells_answer, cells_question);
534
535     fout << difference << "\t";
536 }
537
538 {
539     auto& field = field2;
540     auto cells_answer = grid.makeCells(field, field.startx, field.starty, field.size,
541     ↪ field.size, size, size);
542     auto cells_question = grid.makeCells(field, field.startx, field.starty, field.size,
543     ↪ field.size, size, size);
544     fillWithFunction(cells_answer, f);
545     fillBoundaryConditions3(cells_question, f, 1);
546     auto slae = makeSLAE(cells_question, rightPart);
547     auto answer = solveSLAE(slae);
548     setCells(cells_question, answer);
549     double difference = calcDifference(cells_answer, cells_question);
550
551     fout << difference << std::endl;
552 }
553 }
554 fout.close();
555 }
556
557 int main() {
558     std::vector<NamedFunction> namedFunctions;
559     namedFunctions.push_back({[] (double x, double y) -> double { return 2*x+y; }, "{$2x+y$"});
560     namedFunctions.push_back({[] (double x, double y) -> double { return 3*x*x + y*y + x; },
561     ↪ "{$3x^2+y^2$"});
562     namedFunctions.push_back({[] (double x, double y) -> double { return x*x*x + x*y*y + y*y*y; },
563     ↪ "{$x^3+xy^2+y^3$"});
564     namedFunctions.push_back({[] (double x, double y) -> double { return x*x*x*x + y*y*y*y; },
565     ↪ "{$x^4+y^4$"});
566     namedFunctions.push_back({[] (double x, double y) -> double { return x*x*x*x*x + y*y*y*y*y +
567     ↪ 2*x*y; }, "{$x^5+y^5+2xy$"});
568     namedFunctions.push_back({[] (double x, double y) -> double { return exp(x+y); }, "{$e^{x+y}$"});
569     namedFunctions.push_back({[] (double x, double y) -> double { return exp(x*x+y*y); },
570     ↪ "{$e^{x^2+y^2}$"});
571     namedFunctions.push_back({[] (double x, double y) -> double { return exp(x*x*x+x*x*y); },
572     ↪ "{$e^{x^3+yx^2}$"});
573     namedFunctions.push_back({[] (double x, double y) -> double { return sin(x)+cos(y); }, "{$\sin x +
574     ↪ \cos y$"});
575     namedFunctions.push_back({[] (double x, double y) -> double { return sqrt(x*x+y*y); },
576     ↪ "{$\sqrt{x^2+y^2}$"});
577     namedFunctions.push_back({[] (double x, double y) -> double { return pow(x, 1.2) + pow(y, 1.5); },
578     ↪ "{$x^{1.2}+y^{1.5}$"});
579
580     auto make_tables = [&](std::string add, const Field& field) {
581         std::cout << "Make tables for " << add << std::endl;
582
583         std::cout << "\rWrite condition table" << std::endl;
584         make_table_conditions(add + "conditions.txt", field, namedFunctions);
585
586         std::cout << "\rWrite size table" << std::endl;
587         //make_table_size(add + "size.txt", field, namedFunctions[7].first, 1);
588
589         std::cout << "\rWrite nonuniform grid tables" << std::endl;
590     };
591 }

```

```

575     for (int i = 0; i < namedFunctions.size(); i++) {
576         std::cout << "\r" << i << " " << std::endl;
577         make_table_nonuniform_grid(std::string(add + "non_uniform_grid_") + std::to_string(i) +
578             ↳ std::string(".txt"), field, namedFunctions[i].first, 20, 1);
579     };
580
581     make_tables("0_0_", ShField(0, 0, 1, 1));
582     make_tables("1_2_", ShField(1, 2, 1, 1));
583 }

```

6.2 Код из другой лабораторной работы

6.2.1 Заголовочные файлы

FILE **diagonal.h**

```

1 #pragma once
2
3 #include <string>
4 #include <vector>
5 #include <iostream>
6 #include <map>
7 #include <functional>
8 #include "../1/common.h"
9 #include "../1/vector.h"
10 #include "../1/matrix.h"
11
12 class MatrixDiagonal;
13 class matrix_diagonal_iterator;
14 class SolverSLAE_Iterative;
15
16 //-----
17 /** Класс для вычислений различных параметров с диагональными матрицами. */
18 class Diagonal
19 {
20 public:
21     int n;
22
23     //-----
24     Diagonal(int n);
25
26     int calcDiagonalsCount(void);
27     int calcMinDiagonal(void);
28     int calcMaxDiagonal(void);
29     int calcDiagonalSize(int d);
30
31     bool isLineIntersectDiagonal(int line, int d);
32     bool isRowIntersectDiagonal(int row, int d);
33
34     //-----
35     /**
36      * R - Row - столбец
37      * L - Line - строка
38      * P - Pos - номер элемента в диагонали
39      * D - Diag - формат диагонали
40      */
41
42     int calcLine_byDP(int d, int pos);
43     int calcRow_byDP(int d, int pos);
44
45     int calcDiag_byLR(int line, int row);
46     int calcPos_byLR(int line, int row);
47
48     int calcPos_byDL(int d, int line);
49     int calcPos_byDR(int d, int row);
50
51     int calcRow_byDL(int d, int line);
52     int calcLine_byDR(int d, int row);
53 };
54
55 //-----
56 /** Матрица в диагональном формате. */
57 /** 0-я диагональ всегда главная диагональ. */
58 class MatrixDiagonal
59 {
60 public:
61     typedef std::vector<real>::iterator iterator;
62     typedef std::vector<real>::const_iterator const_iterator;
63
64     //-----
65     MatrixDiagonal();

```

```

66 MatrixDiagonal(int n, std::vector<int> format); // format[0] must be 0, because it's main diagonal
67 MatrixDiagonal(const Matrix& a);
68
69 void toDenseMatrix(Matrix& dense) const;
70 void resize(int n, std::vector<int> format); // format[0] must be 0, because it's main diagonal
71
72 void save(std::ostream& out) const;
73 void load(std::istream& in);
74
75 //-----
76 int dimension(void) const;
77 int getDiagonalsCount(void) const;
78 int getDiagonalSize(int diagNo) const;
79 int getDiagonalPos(int diagNo) const;
80
81 std::vector<int> getFormat(void) const;
82
83 //-----
84 matrix_diagonal_iterator posBegin(int diagNo) const;
85 matrix_diagonal_iterator posEnd(int diagNo) const;
86
87 iterator begin(int diagNo);
88 const_iterator begin(int diagNo) const;
89
90 iterator end(int diagNo);
91 const_iterator end(int diagNo) const;
92
93 private:
94     std::vector<std::vector<real>> di;
95     std::vector<int> fi;
96     int n;
97     Diagonal dc;
98 };
99
100 std::vector<int> makeSevenDiagonalFormat(int n, int m, int k);
101 std::vector<int> generateRandomFormat(int n, int diagonalsCount);
102
103 void generateDiagonalMatrix(
104     int n,
105     int min, int max,
106     std::vector<int> format,
107     MatrixDiagonal& result
108 );
109
110 void generateDiagonallyDominantMatrix(
111     int n,
112     std::vector<int> format,
113     bool isNegative,
114     MatrixDiagonal& result
115 );
116
117 bool mul(const MatrixDiagonal& a, const Vector& x, Vector& y);
118
119 //-----
120 /** Матричный "итератор" для движения по диагонали. */
121 class matrix_diagonal_iterator
122 {
123 public:
124     matrix_diagonal_iterator(int n, int d, bool isEnd);
125
126     matrix_diagonal_iterator& operator++();
127     matrix_diagonal_iterator operator++(int);
128
129     bool operator==(const matrix_diagonal_iterator& b) const;
130     bool operator!=(const matrix_diagonal_iterator& b) const;
131
132     matrix_diagonal_iterator& operator+=(const ptrdiff_t& movement);
133
134     int i, j;
135 };
136
137 /** Матричный "итератор" для движения по строке между различными диагоналями. Может обрабатывать как
138 ↪ всю строку, так и только нижний треугольник. */
139 class matrix_diagonal_line_iterator
140 {
141 public:
142     matrix_diagonal_line_iterator(int n, std::vector<int> format, bool isOnlyLowTriangle);
143
144     matrix_diagonal_line_iterator& operator++();
145     matrix_diagonal_line_iterator operator++(int);
146
147     bool isLineEnd(void) const;
148     bool isEnd(void) const;
149
150     int i, j; // i - текущая строка, j - текущий столбец
151     int d, dn, di; // d - формат текущей диагонали, d - номер текущей диагонали, di - номер текущего
152     ↪ элемента в диагонали
153 private:
154     std::map<int, int> m_map;

```



```

153     std::vector<int> m_sorted_format;
154
155     int line, start, end, pos;
156     Diagonal dc;
157
158     bool m_isLineEnd;
159     bool m_isEnd;
160
161     void calcPos(void);
162 };
163
164 //-----
165 /** Класс итеративного решателя СЛАУ для диагональной матрицы. */
166 struct IterationsResult
167 {
168     int iterations;
169     double relativeResidual;
170 };
171
172 class SolverSLAE_Iterative
173 {
174 public:
175     SolverSLAE_Iterative();
176
177     void save(std::ostream& out) const;
178     void load(std::istream& in);
179
180     IterationsResult jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x) const;
181     IterationsResult seidel(const MatrixDiagonal& a, const Vector& y, Vector& x) const;
182
183     double          w;
184     bool            isLog;
185     std::ostream&   log;
186     Vector          start;
187     double          epsilon;
188     int             maxIterations;
189
190 private:
191     mutable Vector x1;
192
193     // До итерации: x - текущее решение. После итерации x - следующее решение.
194     void iteration_jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x) const;
195     void iteration_seidel(const MatrixDiagonal& a, const Vector& y, Vector& x) const;
196
197     typedef std::function<void(const SolverSLAE_Iterative*, const MatrixDiagonal&, const Vector&,
198                               ↪ Vector&)> step_function;
199
200     IterationsResult iteration_process(
201         const MatrixDiagonal& a,
202         const Vector& y,
203         Vector& x,
204         step_function step
205     ) const;
206 };

```

FILE vector.h

```

1 #pragma once
2
3 #include <string>
4 #include <vector>
5 #include "matrix.h"
6 #include "common.h"
7
8 //-----
9 class Vector
10 {
11 public:
12     Vector();
13     Vector(int size, real fill = 0);
14     Vector(const Matrix& a);
15
16     void loadFromFile(std::string fileName);
17     void saveToFile(std::string fileName) const;
18
19     void save(std::ostream& out) const;
20     void load(std::istream& in);
21
22     void generate(int n, int min, int max);
23     void generate(int n);
24
25     //-----
26     void resize(int n, real fill = 0);
27
28     void negate(void);
29

```

```

30 void zero(void);
31
32 void toDenseMatrix(Matrix& dense, bool isVertical) const;
33
34 //-----
35 int size(void) const;
36
37 real& operator()(int i);
38 const real& operator()(int i) const;
39
40 private:
41     std::vector<real> mas;
42 };
43
44 //-----
45 sumreal sumAllElementsAbs(const Vector& a);
46 bool sum(const Vector& a, const Vector& b, Vector& result);
47 bool mul(const Matrix& a, const Vector& b, Vector& result);
48 real calcNorm(const Vector& a);

```

FILE common.h

```

1 #pragma once
2
3 #ifdef ALL_FLOAT
4     typedef float real;
5     typedef float sumreal;
6 #endif
7
8 #ifdef ALL_DOUBLE
9     typedef double real;
10    typedef double sumreal;
11 #endif
12
13 #ifdef ALL_FLOAT_WITH_DOUBLE
14     typedef float real;
15     typedef double sumreal;
16 #endif
17
18 #ifndef ALL_FLOAT
19 #ifndef ALL_DOUBLE
20 #ifndef ALL_FLOAT_WITH_DOUBLE
21     #error "Type isn't defined"
22     typedef double real;
23     typedef double sumreal;
24 #endif
25 #endif
26 #endif
27
28 bool isNear(double a, double b);
29 double random(void);
30 int intRandom(int min, int max);

```

FILE matrix.h

```

1 #pragma once
2
3 #include <string>
4 #include <vector>
5 #include "common.h"
6
7 //-----
8 class Matrix
9 {
10 public:
11     Matrix(int n = 0, int m = 0, real fill = 0); // n - количество столбцов, m - количество строк
12     void loadFromFile(std::string fileName);
13     void saveToFile(std::string fileName) const;
14
15     void save(std::ostream& out) const;
16     void load(std::istream& in);
17
18     void getFromVector(int n, int m, const std::vector<real>& data);
19
20     void resize(int n, int m, real fill = 0);
21     void negate(void);
22
23     bool isSymmetric(void) const;
24     bool isLowerTriangular(void) const;
25     bool isUpperTriangular(void) const;
26     bool isDiagonal(void) const;
27     bool isDiagonalIdentity(void) const;
28     bool isDegenerate(void) const;
29
30     real& operator()(int i, int j);

```

```

31     const real& operator()(int i, int j) const;
32
33     int width(void) const;
34     int height(void) const;
35
36 private:
37     std::vector<std::vector<real>> m_matrix;
38     int m_n, m_m;
39 };
40
41 //-----
42 void generateSparseSymmetricMatrix(
43     int n,
44     int min, int max,
45     real percent,
46     Matrix& result
47 );
48
49 void generateLMatrix(
50     int n,
51     int min, int max,
52     real percent,
53     Matrix& result
54 );
55
56 void generateDiagonalMatrix(
57     int n,
58     int min, int max,
59     Matrix& result
60 );
61
62 void generateVector(
63     int n,
64     int min, int max,
65     Matrix& result
66 );
67
68 void generateVector(
69     int n,
70     Matrix& result
71 );
72
73 void generateGilbertMatrix(int n, Matrix& result);
74
75 void generateTestMatrix(int n, int profileSize, Matrix& result);
76
77 //-----
78 bool mul(const Matrix& a, const Matrix& b, Matrix& result);
79 bool sum(const Matrix& a, const Matrix& b, Matrix& result);
80
81 sumreal sumAllElementsAbs(const Matrix& a);
82
83 bool transpose(Matrix& a);
84
85 bool calcLDL(const Matrix& a, Matrix& l, Matrix& d);
86 bool calcLUsq(const Matrix& a, Matrix& l, Matrix& u);
87 bool calcLUsq_partial(const Matrix& a, Matrix& l, Matrix& u);
88 bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x);
89 bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x);
90 bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x);
91 bool solveSLAE_by_LDL(const Matrix& a, const Matrix& y, Matrix& x);
92
93 bool solveSLAE_byGaussMethod(const Matrix& a, const Matrix& y, Matrix& x);

```

6.2.2 Файлы исходного кода

FILE diagonal.cpp

```

1 #include <cmath>
2 #include <iostream>
3 #include <iomanip>
4 #include <algorithm>
5 #include "diagonal.h"
6
7 //-----
8 Diagonal::Diagonal(int n) : n(n) {
9 }
10
11 //-----
12 int Diagonal::calcDiagonalsCount(void) {
13     return 2 * n - 1;
14 }
15

```

```

16 //-----
17 int Diagonal::calcMinDiagonal(void) {
18     return -(n-1);
19 }
20
21 //-----
22 int Diagonal::calcMaxDiagonal(void) {
23     return n - 1;
24 }
25
26 //-----
27 int Diagonal::calcDiagonalSize(int d) {
28     return n - std::abs(d);
29 }
30
31 //-----
32 bool Diagonal::isLineIntersectDiagonal(int line, int d) {
33     if (d <= 0)
34         return (line+d >= 0);
35
36     if (d > 0)
37         return (line < calcDiagonalSize(d));
38 }
39
40 //-----
41 bool Diagonal::isRowIntersectDiagonal(int row, int d) {
42     return isLineIntersectDiagonal(row, -d);
43 }
44
45 //-----
46 int Diagonal::calcLine_byDP(int d, int pos) {
47     if (d <= 0)
48         return -d + pos;
49
50     if (d > 0)
51         return pos;
52 }
53
54 //-----
55 int Diagonal::calcRow_byDP(int d, int pos) {
56     if (d <= 0)
57         return pos;
58
59     if (d > 0)
60         return pos + d;
61 }
62
63 //-----
64 int Diagonal::calcDiag_byLR(int line, int row) {
65     return row - line;
66 }
67
68 //-----
69 int Diagonal::calcPos_byLR(int line, int row) {
70     return calcPos_byDL(calcDiag_byLR(line, row), line);
71 }
72
73 //-----
74 int Diagonal::calcPos_byDL(int d, int line) {
75     if (d <= 0)
76         return line+d;
77
78     if (d > 0)
79         return line;
80 }
81
82 //-----
83 int Diagonal::calcPos_byDR(int d, int row) {
84     return calcPos_byDL(d, calcLine_byDR(d, row));
85 }
86
87 //-----
88 int Diagonal::calcRow_byDL(int d, int line) {
89     return line+d;
90 }
91
92 //-----
93 int Diagonal::calcLine_byDR(int d, int row) {
94     return calcRow_byDL(-d, row);
95 }
96
97 //-----
98 //-----
99 //-----
100
101 //-----
102 MatrixDiagonal::MatrixDiagonal() : n(0), dc(n) {
103 }
104

```

```

105 //-----
106 MatrixDiagonal::MatrixDiagonal(int n, std::vector<int> format) : dc(n) {
107     resize(n, format);
108 }
109
110 //-----
111 MatrixDiagonal::MatrixDiagonal(const Matrix& a) : dc(n) {
112     if (a.width() != a.height())
113         throw std::exception();
114
115     n = a.width();
116     dc.n = n;
117
118     // Определяем формат
119     std::vector<int> format;
120     format.clear();
121     format.push_back(0);
122     for (int i = dc.calcMinDiagonal(); i <= dc.calcMaxDiagonal(); ++i)
123         if (i != 0) {
124             auto mit = matrix_diagonal_iterator(n, i, false);
125             auto mite = matrix_diagonal_iterator(n, i, true);
126             for (; mit != mite; ++mit) {
127                 if (a(mit.i, mit.j) != 0) {
128                     format.push_back(i);
129                     break;
130                 }
131             }
132         }
133
134     // Создаем формат
135     resize(n, format);
136
137     // Обходим массив и записываем элементы
138     for (int i = 0; i < getDiagonalsCount(); ++i) {
139         auto mit = posBegin(i);
140         for (auto it = begin(i); it != end(i); ++it, ++mit)
141             *it = a(mit.i, mit.j);
142     }
143 }
144
145 //-----
146 void MatrixDiagonal::toDenseMatrix(Matrix& dense) const {
147     dense.resize(n, 0);
148
149     // Обходим массив и записываем элементы
150     for (int i = 0; i < getDiagonalsCount(); ++i) {
151         auto mit = posBegin(i);
152         for (auto it = begin(i); it != end(i); ++it, ++mit)
153             dense(mit.i, mit.j) = *it;
154     }
155 }
156
157 //-----
158 void MatrixDiagonal::resize(int n1, std::vector<int> format) {
159     if (format[0] != 0)
160         throw std::exception();
161
162     dc.n = n1;
163     n = n1;
164     fi = format;
165
166     di.clear();
167     for (const auto& i : format)
168         di.push_back(std::vector<real>(dc.calcDiagonalSize(i), 0));
169 }
170
171 //-----
172 void MatrixDiagonal::save(std::ostream& out) const {
173     out << n << " " << fi.size() << std::endl;
174     for (const auto& i : fi)
175         out << i << " ";
176     out << std::endl;
177
178     for (int i = 0; i < getDiagonalsCount(); ++i) {
179         for (auto j = begin(i); j != end(i); ++j) {
180             out << (*j) << " ";
181         }
182         out << std::endl;
183     }
184     out << std::endl;
185 }
186
187 //-----
188 void MatrixDiagonal::load(std::istream& in) {
189     int n, m;
190     in >> n >> m;
191     std::vector<int> format(m, 0);
192     for (int i = 0; i < m; ++i)
193         in >> format[i];

```

```

194     resize(n, format);
195     for (int i = 0; i < getDiagonalsCount(); ++i) {
196         for (auto j = begin(i); j != end(i); ++j) {
197             in >> (*j);
198         }
199     }
200 }
201
202 //-----
203 int MatrixDiagonal::dimension(void) const {
204     return n;
205 }
206
207 //-----
208 int MatrixDiagonal::getDiagonalsCount(void) const {
209     return di.size();
210 }
211
212 //-----
213 int MatrixDiagonal::getDiagonalSize(int diagNo) const {
214     return di[diagNo].size();
215 }
216
217 //-----
218 int MatrixDiagonal::getDiagonalPos(int diagNo) const {
219     return fi[diagNo];
220 }
221
222 //-----
223 std::vector<int> MatrixDiagonal::getFormat(void) const {
224     return fi;
225 }
226
227 //-----
228 matrix_diagonal_iterator MatrixDiagonal::posBegin(int diagNo) const {
229     return matrix_diagonal_iterator(n, fi[diagNo], false);
230 }
231
232 //-----
233 matrix_diagonal_iterator MatrixDiagonal::posEnd(int diagNo) const {
234     return matrix_diagonal_iterator(n, fi[diagNo], true);
235 }
236
237 //-----
238 MatrixDiagonal::iterator MatrixDiagonal::begin(int diagNo) {
239     return di[diagNo].begin();
240 }
241
242 //-----
243 MatrixDiagonal::const_iterator MatrixDiagonal::begin(int diagNo) const {
244     return di[diagNo].begin();
245 }
246
247 //-----
248 MatrixDiagonal::iterator MatrixDiagonal::end(int diagNo) {
249     return di[diagNo].end();
250 }
251
252 //-----
253 MatrixDiagonal::const_iterator MatrixDiagonal::end(int diagNo) const {
254     return di[diagNo].end();
255 }
256
257 //-----
258 //-----
259 //-----
260
261 //-----
262 matrix_diagonal_iterator::matrix_diagonal_iterator(int n, int d, bool isEnd) {
263     Diagonal dc(n);
264     if (isEnd) {
265         i = dc.calcLine_byDP(d, dc.calcDiagonalSize(d));
266         j = dc.calcRow_byDP(d, dc.calcDiagonalSize(d));
267     } else {
268         i = dc.calcLine_byDP(d, 0);
269         j = dc.calcRow_byDP(d, 0);
270     }
271 }
272
273 //-----
274 matrix_diagonal_iterator& matrix_diagonal_iterator::operator++() {
275     i++;
276     j++;
277     return *this;
278 }
279
280 //-----
281 matrix_diagonal_iterator matrix_diagonal_iterator::operator++(int) {
282     i++;

```

```

283     j++;
284     return *this;
285 }
286
287 //-----
288 bool matrix_diagonal_iterator::operator==(const matrix_diagonal_iterator& b) const {
289     return b.i == i && b.j == j;
290 }
291
292 //-----
293 bool matrix_diagonal_iterator::operator!=(const matrix_diagonal_iterator& b) const {
294     return b.i != i || b.j != j;
295 }
296
297 //-----
298 matrix_diagonal_iterator& matrix_diagonal_iterator::operator+=(const ptrdiff_t& movement) {
299     i += movement;
300     j += movement;
301     return *this;
302 }
303
304 //-----
305 //-----
306 //-----
307
308 //-----
309 matrix_diagonal_line_iterator::matrix_diagonal_line_iterator(int n, std::vector<int> format, bool
↵ isOnlyLowTriangle) : dc(n), m_isEnd(false), m_isLineEnd(false) {
310     // Создаем обратное преобразование из формата диагонали в ее номер в формате
311     for (int i = 0; i < format.size(); ++i)
312         if ((isOnlyLowTriangle && format[i] < 0) || !isOnlyLowTriangle)
313             m_map[format[i]] = i;
314
315     // Создаем сортированный формат, чтобы по нему двигаться
316     if (isOnlyLowTriangle) {
317         for (int i = 0; i < format.size(); ++i)
318             if (format[i] < 0)
319                 m_sorted_format.push_back(format[i]);
320     } else
321         m_sorted_format = format;
322     std::Sort(m_sorted_format.begin(), m_sorted_format.end());
323
324     line = 0;
325     pos = 0;
326     start = m_sorted_format.size() - 1;
327     end = start;
328
329     // Находим, с какой диагонали начинается текущая строка
330     for (int i = 0; i < m_sorted_format.size(); ++i) {
331         if (dc.isLineIntersectDiagonal(line, m_sorted_format[i])) {
332             start = i;
333             break;
334         }
335     }
336
337     // Находим на какой диагонали кончается текущая строка
338     for (int i = 0; i < m_sorted_format.size(); ++i) {
339         int j = m_sorted_format.size() - i - 1;
340         if (dc.isLineIntersectDiagonal(line, m_sorted_format[j])) {
341             end = j;
342             break;
343         }
344     }
345
346     calcPos();
347 }
348
349 //-----
350 matrix_diagonal_line_iterator& matrix_diagonal_line_iterator::operator++() {
351     if (!m_isEnd) {
352         if (m_isLineEnd) {
353             // Сдвигаемся на одну строку
354             line++;
355
356             // Определяем какие диагонали пересекают эту строку
357             if (start != 0)
358                 if (dc.isLineIntersectDiagonal(line, m_sorted_format[start-1]))
359                     start = start-1;
360
361             if (end != 0)
362                 if (!dc.isLineIntersectDiagonal(line, m_sorted_format[end]))
363                     if (start != end)
364                         end = end-1;
365
366             m_isLineEnd = false;
367             if (line == dc.n)
368                 m_isEnd = true;
369
370             pos = 0;

```

```

371         calcPos();
372     } else {
373         // Сдвигаемся на один столбец
374         pos++;
375         calcPos();
376     }
377 }
378
379 return *this;
380 }
381
382 //-----
383 matrix_diagonal_line_iterator matrix_diagonal_line_iterator::operator++(int) {
384     return operator++();
385 }
386
387 //-----
388 bool matrix_diagonal_line_iterator::isLineEnd(void) const {
389     return m_isLineEnd;
390 }
391
392 //-----
393 bool matrix_diagonal_line_iterator::isEnd(void) const {
394     return m_isEnd;
395 }
396
397 //-----
398 void matrix_diagonal_line_iterator::calcPos(void) {
399     // Вычисляет все текущие положения согласно переменным start, pos и формату
400     if (!dc.isLineIntersectDiagonal(line, m_sorted_format[end]) || (start + pos > end)) {
401         m_isLineEnd = true;
402         i = line;
403         j = 0;
404         d = 0;
405         di = 0;
406         dn = 0;
407     } else {
408         i = line;
409         d = m_sorted_format[start + pos];
410         dn = m_map[d];
411         di = dc.calcPos_byDL(d, i);
412         j = dc.calcRow_byDL(d, i);
413     }
414 }
415
416 //-----
417 //-----
418 //-----
419
420 //-----
421 std::vector<int> makeSevenDiagonalFormat(int n, int m, int k) {
422     std::vector<int> result;
423
424     if (1+m+k >= n)
425         throw std::exception();
426
427     result.push_back(0);
428
429     result.push_back(1);
430     result.push_back(1+m);
431     result.push_back(1+m+k);
432
433     result.push_back(-1);
434     result.push_back(-1-m);
435     result.push_back(-1-m-k);
436
437     return result;
438 }
439
440 //-----
441 std::vector<int> generateRandomFormat(int n, int diagonalsCount) {
442     Diagonal d(n);
443
444     std::vector<int> result;
445     result.push_back(0);
446
447     // Создаем массив всех возможных диагоналей
448     std::vector<int> diagonals;
449     for (int i = d.calcMinDiagonal(); i <= d.calcMaxDiagonal(); ++i)
450         if (i != 0)
451             diagonals.push_back(i);
452
453     diagonalsCount = std::min<int>(diagonals.size(), diagonalsCount);
454
455     // Заполняем результат случайными диагоналями из этого массива
456     for (int i = 0; i < diagonalsCount; ++i) {
457         int pos = intRandom(0, diagonals.size());
458         result.push_back(diagonals[pos]);
459         diagonals.erase(diagonals.begin() + pos);
460     }
461 }

```



```

460     }
461     return result;
462 }
463 }
464
465 //-----
466 void generateDiagonalMatrix(int n, int min, int max, std::vector<int> format, MatrixDiagonal& result) {
467     result.resize(n, format);
468     for (int i = 0; i < result.getDiagonalsCount(); ++i) {
469         auto mit = result.posBegin(i);
470         for (auto it = result.begin(i); it != result.end(i); ++it, ++mit)
471             (*it) = intRandom(min, max);
472     }
473 }
474
475 //-----
476 void generateDiagonallyDominantMatrix(int n, std::vector<int> format, bool isNegative,
477 ↪ MatrixDiagonal& result) {
478     result.resize(n, format);
479
480     for (int i = 0; i < result.getDiagonalsCount(); ++i) {
481         auto mit = result.posBegin(i);
482         for (auto it = result.begin(i); it != result.end(i); ++it, ++mit) {
483             if (isNegative)
484                 *it = -intRandom(0, 5);
485             else
486                 *it = intRandom(0, 5);
487         }
488     }
489
490     matrix_diagonal_line_iterator mit(n, format, false);
491     for (; !mit.isEnd(); ++mit) {
492         sumreal& sum = result.begin(0)[mit.i];
493         sum = 0;
494         for (; !mit.isLineEnd(); ++mit)
495             if (mit.i != mit.j)
496                 sum += result.begin(mit.dn)[mit.di];
497         sum = std::fabs(sum);
498     }
499     result.begin(0)[0] += 1;
500 }
501
502 //-----
503 bool mul(const MatrixDiagonal& a, const Vector& x, Vector& y) {
504     if (x.size() != a.dimension())
505         return false;
506
507     y.resize(x.size());
508
509     // Зануление результата
510     y.zero();
511     for (int i = 0; i < a.getDiagonalsCount(); ++i) {
512         auto mit = a.posBegin(i);
513         for (auto it = a.begin(i); it != a.end(i); ++it, ++mit)
514             y(mit.i) += (*it) * x(mit.j);
515     }
516
517     return true;
518 }
519
520 //-----
521 //-----
522 //-----
523
524 //-----
525 SolverSLAE_Iterative::SolverSLAE_Iterative() :
526     w(1),
527     isLog(false),
528     log(std::cout),
529     start(),
530     epsilon(0.00001),
531     maxIterations(100) {
532 }
533
534 //-----
535 void SolverSLAE_Iterative::save(std::ostream& out) const {
536     out << w << std::endl;
537     out << isLog << std::endl;
538     start.save(out);
539     out << std::scientific;
540     out << epsilon << std::endl;
541     out << std::defaultfloat;
542     out << maxIterations << std::endl;
543 }
544
545 //-----
546 void SolverSLAE_Iterative::load(std::istream& in) {

```

```

547     in >> w >> isLog;
548     start.load(in);
549     in >> epsilon >> maxIterations;
550 }
551
552 //-----
553 IterationsResult SolverSLAE_Iterative::jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x)
554 ↪ const {
555     return iteration_process(a, y, x, &SolverSLAE_Iterative::iteration_jacobi);
556 }
557
558 //-----
559 IterationsResult SolverSLAE_Iterative::seidel(const MatrixDiagonal& a, const Vector& y, Vector& x)
560 ↪ const {
561     return iteration_process(a, y, x, &SolverSLAE_Iterative::iteration_seidel);
562 }
563
564 //-----
565 IterationsResult SolverSLAE_Iterative::iteration_process(const MatrixDiagonal& a, const Vector& y,
566 ↪ Vector& x, step_function step) const {
567     if (a.dimension() != y.size() || start.size() != y.size())
568         throw std::exception();
569
570     // Считаем норму матрицы: ее максимальный элемент по модулю
571     real yNorm = calcNorm(y);
572     x1.resize(y.size());
573     x = start;
574
575     // Цикл по итерациям
576     int i = 0;
577     real relativeResidual = epsilon + 1;
578     for (; i < maxIterations && relativeResidual > epsilon; ++i) {
579         // Итерационный шаг
580         step(this, a, y, x);
581
582         // Считаем невязку
583         mul(a, x, x1);
584         x1.negate();
585         sum(x1, y, x1);
586         relativeResidual = fabs(calcNorm(x1)) / yNorm;
587
588         // Выводим данные
589         if (isLog)
590             log << i << "\t" << std::scientific << std::setprecision(3) << relativeResidual <<
591             ↪ std::endl;
592     }
593     return {i, relativeResidual};
594 }
595
596 //-----
597 void SolverSLAE_Iterative::iteration_jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x) const
598 {
599     // Умножаем матрицу на решение
600     mul(a, x, x1);
601
602     //  $x^{(k+1)} = x^{(k)} + w/a(i, i) * x^{(k+1)}$ 
603     auto it = a.begin(0);
604     for (int i = 0; i < x1.size(); ++i, ++it)
605         x(i) += w / (*it) * (y(i) - x1(i));
606 }
607
608 //-----
609 void SolverSLAE_Iterative::iteration_seidel(const MatrixDiagonal& a, const Vector& y, Vector& x) const
610 {
611     // Умножим верхний треугольник на решение
612     x1.zero();
613     for (int i = 0; i < a.getDiagonalsCount(); ++i)
614         if (a.getDiagonalPos(i) >= 0) {
615             auto mit = a.posBegin(i);
616             for (auto it = a.begin(i); it != a.end(i); ++it, ++mit)
617                 x1(mit.i) += (*it) * x(mit.j);
618         }
619
620     // Проходим по нижнему треугольнику и считаем все параметры
621     matrix_diagonal_line_iterator mit(a.dimension(), a.getFormat(), true);
622     for (; !mit.isEnd(); ++mit) {
623         for (; !mit.isLineEnd(); ++mit)
624             x1(mit.i) += a.begin(mit.dn)[mit.di] * x(mit.j);
625         x(mit.i) = x(mit.i) + w/a.begin(0)[mit.i] * (y(mit.i) - x1(mit.i));
626     }
627 }

```

```

1 #include <fstream>
2 #include <iomanip>
3 #include <cmath>
4 #include "vector.h"
5
6 //-----
7 Vector::Vector() {
8 }
9
10 //-----
11 Vector::Vector(int size, real fill) : mas(size, fill) {
12 }
13
14 //-----
15 Vector::Vector(const Matrix& a) {
16     if (a.width() == 1) {
17         resize(a.height());
18         for (int i = 0; i < a.height(); ++i)
19             mas[i] = a(i, 0);
20     } else if (a.height() == 1) {
21         resize(a.width());
22         for (int i = 0; i < a.width(); ++i)
23             mas[i] = a(0, i);
24     } else
25         throw std::exception();
26 }
27
28 //-----
29 void Vector::toDenseMatrix(Matrix& dense, bool isVertical) const {
30     if (isVertical) {
31         dense.resize(1, mas.size());
32         for (int i = 0; i < mas.size(); ++i)
33             dense(i, 0) = mas[i];
34     } else {
35         dense.resize(mas.size(), 1);
36         for (int i = 0; i < mas.size(); ++i)
37             dense(0, i) = mas[i];
38     }
39 }
40
41 //-----
42 void Vector::loadFromFile(std::string fileName) {
43     std::ifstream fin(fileName);
44     load(fin);
45     fin.close();
46 }
47
48 //-----
49 void Vector::saveToFile(std::string fileName) const {
50     std::ofstream fout(fileName);
51     fout.precision(std::numeric_limits<real>::digits10);
52     save(fout);
53     fout.close();
54 }
55
56 //-----
57 void Vector::save(std::ostream& out) const {
58     out << mas.size() << std::endl;
59     for (const auto& i : mas)
60         out << i << std::endl;
61     out << std::endl;
62 }
63
64 //-----
65 void Vector::load(std::istream& in) {
66     int n;
67     in >> n;
68     resize(n);
69     for (int i = 0; i < n; ++i)
70         in >> mas[i];
71 }
72
73 //-----
74 void Vector::resize(int n, real fill) {
75     if (mas.size() != n)
76         mas.resize(n, fill);
77 }
78
79 //-----
80 void Vector::negate(void) {
81     for (auto& i : mas)
82         i = -i;
83 }
84
85 //-----
86 void Vector::zero(void) {
87     for (auto& i : mas)

```

```

88         i = 0;
89     }
90
91     //-----
92     void Vector::generate(int n, int min, int max) {
93         resize(n);
94         for (int i = 0; i < n; ++i)
95             operator()(i) = intRandom(min, max);
96     }
97
98     //-----
99     void Vector::generate(int n) {
100         resize(n);
101         for (int i = 0; i < n; ++i)
102             operator()(i) = i+1;
103     }
104
105     //-----
106     int Vector::size(void) const {
107         return mas.size();
108     }
109
110     //-----
111     real& Vector::operator()(int i) {
112         return mas[i];
113     }
114
115     //-----
116     const real& Vector::operator()(int i) const {
117         return mas[i];
118     }
119
120     //-----
121     //-----
122     //-----
123
124     //-----
125     bool sum(const Vector& a, const Vector& b, Vector& result) {
126         if (a.size() != b.size())
127             return false;
128
129         if (result.size() != a.size())
130             result.resize(a.size());
131
132         for (int i = 0; i < result.size(); ++i)
133             result(i) = a(i) + b(i);
134
135         return true;
136     }
137
138     //-----
139     sumreal sumAllElementsAbs(const Vector& a) {
140         sumreal sum = 0;
141         for (int i = 0; i < a.size(); ++i)
142             sum += fabs(a(i));
143
144         return sum;
145     }
146
147     //-----
148     bool mul(const Matrix& a, const Vector& b, Vector& result) {
149         // result = a * b
150         if (a.width() != b.size())
151             return false;
152
153         result.resize(b.size());
154
155         for (int i = 0; i < a.height(); ++i) {
156             sumreal sum = 0;
157             for (int j = 0; j < a.width(); ++j)
158                 sum += a(i, j) * b(j);
159             result(i) = sum;
160         }
161
162         return true;
163     }
164
165     //-----
166     real calcNorm(const Vector& a) {
167         sumreal sum = 0;
168         for (int i = 0; i < a.size(); ++i)
169             sum += a(i) * a(i);
170
171         return std::sqrt(sum);
172     }

```

FILE common.cpp

```

1 #include <cmath>
2 #include "common.h"
3
4 //-----
5 bool isNear(double a, double b) {
6     if (a != 0) {
7         if (fabs(a - b)/a > 0.0001)
8             return false;
9     } else {
10         if (fabs(b) > 0.0001)
11             return false;
12     }
13
14     return true;
15 }
16
17 //-----
18 double random(void) {
19     return std::rand() / double(RAND_MAX);
20 }
21
22 //-----
23 int intRandom(int min, int max) {
24     return min + random() * (max - min);
25 }

```

FILE matrix.cpp

```

1 #include <fstream>
2 #include <iomanip>
3 #include "matrix.h"
4
5 //-----
6 Matrix::Matrix(int n, int m, real fill) : m_matrix(m, std::vector<real>(n, fill)), m_n(n), m_m(m) {
7 }
8
9 //-----
10 void Matrix::loadFromFile(std::string fileName) {
11     std::ifstream fin(fileName);
12
13     m_matrix.clear();
14     int n, m;
15     fin >> n >> m;
16     resize(n, m);
17     for (int i = 0; i < height(); ++i) {
18         for (int j = 0; j < width(); ++j) {
19             fin >> operator()(i, j);
20         }
21     }
22
23     fin.close();
24 }
25
26 //-----
27 void Matrix::saveToFile(std::string fileName) const {
28     std::ofstream fout(fileName);
29
30     fout.precision(std::numeric_limits<real>::digits10);
31     int w = std::numeric_limits<real>::digits10 + 4;
32     save(fout);
33
34     fout.close();
35 }
36
37 //-----
38 void Matrix::load(std::istream& in) {
39     m_matrix.clear(); m_m = 0; m_n = 0;
40     int n, m;
41     in >> n >> m;
42     resize(n, m);
43     for (int i = 0; i < height(); ++i) {
44         for (int j = 0; j < width(); ++j) {
45             in >> operator()(i, j);
46         }
47     }
48 }
49
50 //-----
51 void Matrix::save(std::ostream& out) const {
52     out << m_n << "\t" << m_m << std::endl;
53     for (int i = 0; i < height(); ++i) {
54         for (int j = 0; j < width(); ++j)
55             out << "\t" << std::setw(10) << operator()(i, j);
56         out << std::endl;
57     }
58 }

```

```

57     }
58     out << std::endl;
59 }
60
61 //-----
62 void Matrix::getFromVector(int n, int m, const std::vector<real>& data) {
63     resize(n, m);
64     for (int i = 0; i < data.size(); ++i)
65         operator()(i / n, i % n) = data[i];
66 }
67
68 //-----
69 void Matrix::resize(int n, int m, real fill) {
70     if (m_n != n || m_m != m) {
71         m_n = n;
72         m_m = m;
73         m_matrix.clear();
74         m_matrix.resize(m_m, std::vector<real>(m_n, fill));
75     }
76 }
77
78 //-----
79 void Matrix::negate(void) {
80     for (auto& i : m_matrix) {
81         for (auto& j : i) {
82             j = -j;
83         }
84     }
85 }
86
87 //-----
88 bool Matrix::isSymmetric(void) const {
89     if (height() != width())
90         return false;
91
92     for (int i = 0; i < height(); ++i) {
93         for (int j = 0; j <= i; ++j) {
94             const real& a = operator()(i, j);
95             const real& b = operator()(j, i);
96             if (!isNear(a, b))
97                 return false;
98         }
99     }
100     return true;
101 }
102
103 //-----
104 bool Matrix::isLowerTriangular(void) const {
105     if (height() != width())
106         return false;
107
108     for (int i = 0; i < height(); ++i) {
109         for (int j = 0; j < i; ++j) {
110             if (fabs(operator()(j, i)) > 0.000001)
111                 return false;
112         }
113     }
114     return true;
115 }
116
117 //-----
118 bool Matrix::isUpperTriangular(void) const {
119     if (height() != width())
120         return false;
121
122     for (int i = 0; i < height(); ++i) {
123         for (int j = 0; j < i; ++j) {
124             if (operator()(i, j) != 0)
125                 return false;
126         }
127     }
128     return true;
129 }
130
131 //-----
132 bool Matrix::isDiagonal(void) const {
133     if (height() != width())
134         return false;
135
136     for (int i = 0; i < height(); ++i) {
137         for (int j = 0; j < i; ++j) {
138             if (operator()(j, i) != 0 && operator()(i, j) != 0)
139                 return false;
140         }
141     }
142     return true;
143 }
144
145 //-----

```

```

146     return true;
147 }
148
149 //-----
150 bool Matrix::isDiagonalIdentity(void) const {
151     if (height() != width())
152         return false;
153
154     for (int i = 0; i < height(); ++i) {
155         if (operator()(i, i) != 1)
156             return false;
157     }
158
159     return true;
160 }
161
162 //-----
163 bool Matrix::isDegenerate(void) const {
164     // TODO
165     return false;
166 }
167
168 //-----
169 real& Matrix::operator()(int i, int j) {
170     return m_matrix[i][j];
171 }
172
173 //-----
174 const real& Matrix::operator()(int i, int j) const {
175     return m_matrix[i][j];
176 }
177
178 //-----
179 int Matrix::width(void) const {
180     return m_n;
181 }
182
183 //-----
184 int Matrix::height(void) const {
185     return m_m;
186 }
187
188 //-----
189 //-----
190 //-----
191 //-----
192
193 //-----
194 void generateSparseSymmetricMatrix(int n, int min, int max, real percent, Matrix& result) {
195     result.resize(n, n, 0);
196
197     int count = percent * n * n;
198
199     for (int k = 0; k < count; ++k) {
200         int i = intRandom(0, n-1);
201         int j = intRandom(0, i-1);
202         result(i, j) = intRandom(min, max);
203         result(j, i) = result(i, j);
204     }
205
206     for (int i = 0; i < n; ++i)
207         result(i, i) = intRandom(1, max - min);
208 }
209
210 //-----
211 void generateLMMatrix(int n, int min, int max, real percent, Matrix& result) {
212     result.resize(n, n, 0);
213
214     int count = percent * n * n / 2;
215
216     for (int k = 0; k < count; ++k) {
217         int i = intRandom(0, n-1);
218         int j = intRandom(0, i-1);
219         result(i, j) = intRandom(min, max);
220     }
221
222     for (int i = 0; i < n; ++i) {
223         result(i, i) = 1;
224     }
225 }
226
227 //-----
228 void generateDiagonalMatrix(int n, int min, int max, Matrix& result) {
229     result.resize(n, n, 0);
230
231     for (int i = 0; i < n; ++i)
232         result(i, i) = intRandom(min, max);
233 }
234

```

```

235 //-----
236 void generateVector(int n, int min, int max, Matrix& result) {
237     result.resize(1, n, 0);
238
239     for (int i = 0; i < n; ++i)
240         result(i, 0) = intRandom(min, max);
241 }
242
243 //-----
244 void generateVector(int n, Matrix& result) {
245     result.resize(1, n, 0);
246
247     for (int i = 0; i < n; ++i)
248         result(i, 0) = i+1;
249 }
250
251 //-----
252 void generateGilbertMatrix(int n, Matrix& result) {
253     result.resize(n, n);
254
255     for (int i = 0; i < n; ++i) {
256         for (int j = 0; j < n; ++j) {
257             result(i, j) = double(1.0)/double((i+1)+(j+1)-1);
258         }
259     }
260 }
261
262 //-----
263 void generateTestMatrix(int n, int profileSize, Matrix& result) {
264     result.resize(n, n);
265
266     for (int i = 0; i < n; ++i) {
267         for (int j = 0; j < profileSize; ++j) if (i-j-1 >= 0) {
268             result(i, i-j-1) = -intRandom(0, 5);
269             result(i-j-1, i) = result(i, i-j-1);
270         }
271     }
272
273     for (int i = 0; i < n; ++i) {
274         sumreal sum = 0;
275         for (int j = 0; j < n; ++j) if (i != j) {
276             sum += result(i, j);
277         }
278         result(i, i) = -sum;
279     }
280 }
281
282 //-----
283 //-----
284 //-----
285
286 //-----
287 bool mul(const Matrix& a, const Matrix& b, Matrix& result) {
288     // result = rus_a * b
289     if (a.width() != b.height())
290         return false;
291
292     result.resize(b.width(), a.height());
293
294     for (int i = 0; i < b.width(); ++i) {
295         for (int j = 0; j < a.height(); ++j) {
296             real sum = 0;
297             for (int k = 0; k < a.width(); ++k) {
298                 sum += a(j, k) * b(k, i);
299             }
300             result(j, i) = sum;
301         }
302     }
303
304     return true;
305 }
306
307 //-----
308 bool sum(const Matrix& a, const Matrix& b, Matrix& result) {
309     // result = rus_a + b
310     if (a.width() != b.width() || a.height() != b.height())
311         return false;
312
313     result.resize(a.width(), a.height());
314
315     for (int i = 0; i < a.width(); ++i) {
316         for (int j = 0; j < a.height(); ++j) {
317             result(j, i) = a(j, i) + b(j, i);
318         }
319     }
320
321     return true;
322 }
323

```



```

324 //-----
325 bool transpose(Matrix& a) {
326     // rus_a = rus_a^T
327     Matrix a_t(a.height(), a.width());
328
329     for (int i = 0; i < a.height(); ++i) {
330         for (int j = 0; j < a.width(); ++j) {
331             a_t(j, i) = a(i, j);
332         }
333     }
334
335     a = a_t;
336
337     return true;
338 }
339
340 //-----
341 sumreal sumAllElementsAbs(const Matrix& a) {
342     sumreal sum = 0;
343     for (int i = 0; i < a.height(); ++i) {
344         for (int j = 0; j < a.width(); ++j) {
345             sum += fabs(a(i, j));
346         }
347     }
348
349     return sum;
350 }
351
352 //-----
353 bool calcLUsq(const Matrix& a, Matrix& l, Matrix& u) {
354     if (a.width() != a.height())
355         return false;
356
357     l.resize(a.width(), a.height(), 0);
358     u.resize(a.width(), a.height(), 0);
359
360     for (int i = 0; i < a.height(); ++i) {
361         // Считаем элементы матрицы L
362         for (int j = 0; j < i; ++j) {
363             real sum = 0;
364             for (int k = 0; k < j; ++k)
365                 sum += l(i, k) * u(k, j);
366
367             l(i, j) = (a(i, j) - sum) / l(j, j);
368         }
369
370         // Считаем элементы матрицы U
371         for (int j = 0; j < i; ++j) {
372             real sum = 0;
373             for (int k = 0; k < j; ++k)
374                 sum += l(j, k) * u(k, i);
375
376             u(j, i) = (a(j, i) - sum) / u(j, j);
377         }
378
379         // Считаем диагональный элемент
380         real sum = 0;
381         for (int k = 0; k < i; ++k)
382             sum += l(i, k) * u(k, i);
383
384         l(i, i) = sqrt(a(i, i) - sum);
385         u(i, i) = l(i, i);
386     }
387
388     return true;
389 }
390
391 //-----
392 bool calcLUsq_partial(const Matrix& a, Matrix& l, Matrix& u) {
393     if (a.width() != a.height())
394         return false;
395
396     l.resize(a.width(), a.height(), 0);
397     u.resize(a.width(), a.height(), 0);
398
399     for (int i = 0; i < a.height(); ++i) {
400         // Считаем элементы матрицы L
401         for (int j = 0; j < i; ++j) if (a(i, j) != 0) {
402             real sum = 0;
403             for (int k = 0; k < j; ++k)
404                 sum += l(i, k) * u(k, j);
405
406             l(i, j) = (a(i, j) - sum) / l(j, j);
407         }
408
409         // Считаем элементы матрицы U
410         for (int j = 0; j < i; ++j) if (a(j, i) != 0) {
411             real sum = 0;
412             for (int k = 0; k < j; ++k)

```

```

413         sum += l(j, k) * u(k, i);
414
415         u(j, i) = (a(j, i) - sum) / u(j, j);
416     }
417
418     // Считаем диагональный элемент
419     real sum = 0;
420     for (int k = 0; k < i; ++k)
421         sum += l(i, k) * u(k, i);
422
423     l(i, i) = sqrt(a(i, i) - sum);
424     u(i, i) = l(i, i);
425 }
426
427 return true;
428 }
429
430 //-----
431 bool calcLDL(const Matrix& a, Matrix& l, Matrix& d) {
432     //  $l * d * l^T = \text{rus}_a$ 
433     if (!a.isSymmetric())
434         return false;
435
436     l.resize(a.width(), a.height(), 0);
437     d.resize(a.width(), a.height(), 0);
438
439     for (int i = 0; i < a.height(); ++i) {
440         // Считаем элементы матрицы L
441         for (int j = 0; j < i; ++j) {
442             real sum = 0;
443             for (int k = 0; k < j; ++k)
444                 sum += d(k, k) * l(j, k) * l(i, k);
445
446             if (fabs(d(j, j)) < 0.0001)
447                 l(i, j) = 0;
448             else
449                 l(i, j) = (a(i, j) - sum) / d(j, j);
450         }
451
452         // Считаем диагональный элемент
453         {
454             real sum = 0;
455             for (int j = 0; j < i; ++j)
456                 sum += d(j, j) * l(i, j) * l(i, j);
457             d(i, i) = a(i, i) - sum;
458         }
459     }
460
461     for (int i = 0; i < l.height(); i++)
462         l(i, i) = 1;
463
464     return true;
465 }
466
467 //-----
468 bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x) {
469     //  $l * x = y$ ,  $l$  - нижнетреугольная матрица
470     if (!l.isLowerTriangular() || !l.isDiagonalIdentity())
471         return false;
472
473     x.resize(1, y.height());
474
475     for (int i = x.height() - 1; i >= 0; --i) {
476         real sum = 0;
477         for (int j = i; j < x.height(); ++j)
478             sum += l(j, i) * x(j, 0);
479         x(i, 0) = y(i, 0) - sum;
480     }
481
482     return true;
483 }
484
485 //-----
486 bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x) {
487     //  $l * x = y$ ,  $l$  - верхнетреугольная матрица
488     if (!l.isLowerTriangular() || !l.isDiagonalIdentity())
489         return false;
490
491     x.resize(1, y.height());
492
493     for (int i = 0; i < x.height(); ++i) {
494         real sum = 0;
495         for (int j = 0; j < i; ++j)
496             sum += l(i, j) * x(j, 0);
497         x(i, 0) = y(i, 0) - sum;
498     }
499
500     return true;
501 }

```

```

502 //-----
503 bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x) {
504     // d * x = y, d - диагональная матрица
505     if (!d.isDiagonal())
506         return false;
507
508     x.resize(1, y.height());
509
510     for (int i = 0; i < x.height(); ++i)
511         x(i, 0) = y(i, 0) / d(i, i);
512
513     return true;
514 }
515
516 //-----
517 bool solveSLAE_by_LDL(const Matrix& a, const Matrix& y, Matrix& x) {
518     // rus_a * x = y, rus_a - симметричная матрица
519     if (!(a.width() == a.height() && a.width() == y.height() && !a.isDegenerate()))
520         return false;
521
522     Matrix l, d, z, w;
523
524     if (!calcLDL(a, l, d))
525         return false;
526
527     if (!calcGaussianFrontOrder(l, y, z))
528         return false;
529
530     if (!calcGaussianCentralOrder(d, z, w))
531         return false;
532
533     if (!calcGaussianReverseOrder(l, w, x))
534         return false;
535
536     return true;
537 }
538
539 //-----
540 bool solveSLAE_byGaussMethod(const Matrix& a1, const Matrix& y1, Matrix& x1) {
541     if (!(a1.width() == a1.height() && y1.height() == a1.width() && !a1.isDegenerate()))
542         return false;
543
544     Matrix a(a1);
545     Matrix y(y1);
546
547     for (int i = 0; i < a.height(); ++i) {
548         // Находим максимальный элемент
549         int maxI = i;
550         for (int j = i+1; j < a.height(); ++j)
551             if (fabs(a(j, i)) > fabs(a(maxI, i)))
552                 maxI = j;
553
554         // Переставляем эту строчку с текущей
555         for (int j = i; j < a.width(); ++j)
556             std::swap(a(i, j), a(maxI, j));
557         std::swap(y(i, 0), y(maxI, 0));
558
559         // Перебираем все строчки ниже и отнимаем текущую строчку от них
560         for (int j = i+1; j < a.height(); ++j) {
561             real m = a(j, i) / a(i, i);
562             for (int k = i; k < a.width(); ++k)
563                 a(j, k) -= m * a(i, k);
564             y(j, 0) -= m * y(i, 0);
565         }
566
567         // Делим текущую строку на ее ведущий элемент, чтобы на диагонали были единицы
568         double m = a(i, i);
569         for (int j = i; j < a.width(); ++j)
570             a(i, j) /= m;
571         y(i, 0) /= m;
572     }
573
574     // Считаем обратный ход Гаусса
575     transpose(a);
576     calcGaussianReverseOrder(a, y, x1);
577
578     return true;
579 }
580

```