

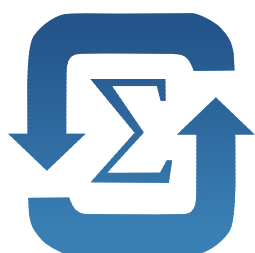
Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



Кафедра прикладной математики

Лабораторная работа №2  
по дисциплине «Численные методы»

## Итерационные методы решения СЛАУ



Факультет:	ПМИ
Группа:	ПМ-63
Студент:	Шепрут И.И.
Вариант:	11
Преподаватель:	Задорожный А.Г.

Новосибирск  
2018

# 1 Цель работы

Разработать программы решения СЛАУ методами Якоби, Гаусса-Зейделя с хранением матрицы в диагональном формате. Исследовать сходимость методов для различных тестовых матриц и её зависимость от параметра релаксации. Изучить возможность оценки порядка числа обусловленности матрицы путем вычислительного эксперимента.

**Вариант 11:** 7-ми диагональная матрица с параметрами  $m$ ,  $k$  — количество нулевых диагоналей,  $n$  — размерность матрицы.

# 2 Код программы

Программа состоит из нескольких частей:

1. `common.h` + `common.cpp` — пара общих функций и объявление вещественных типов.
2. `matrix.h` + `matrix.cpp` — модуль для работы с матрицами в плотном формате.
3. `diagonal.h` + `diagonal.cpp` — модуль для работы с матрицами в диагональном формате.
4. `table_generator.cpp` — программа, которая генерирует таблицы.

## 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
29 bool isNear(double a, double b);
30 double random(void);
31 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 getFromVector(int n, int m, const std::vector<real>& data);
16
17     void resize(int n, int m, real fill = 0);
18     void negate(void);
19
20     bool isSymmetric(void) const;
21     bool isLowerTriangular(void) const;
22     bool isUpperTriangular(void) const;
23     bool isDiagonal(void) const;
24     bool isDiagonalIdentity(void) const;
25     bool isDegenerate(void) const;
26
27     real& operator()(int i, int j);
28     const real& operator()(int i, int j) const;
29
30     int width(void) const;
31     int height(void) const;
32
33 private:
34     std::vector<std::vector<real>> m_matrix;
35     int m_n, m_m;
36 };
37
38 //-----
39 void generateSparseSymmetricMatrix(
40     int n,
41     int min, int max,
42     real percent,
43     Matrix& result
44 );
45
46 void generateLMatrix(
47     int n,
48     int min, int max,
49     real percent,
50     Matrix& result
51 );
52
53 void generateDiagonalMatrix(
54     int n,
55     int min, int max,
56     Matrix& result
57 );
58
59 void generateVector(
60     int n,
61     int min, int max,
62     Matrix& result
63 );

```

```

64
65 void generateVector(
66     int n,
67     Matrix& result
68 );
69
70 void generateGilbertMatrix(int n, Matrix& result);
71
72 void generateTestMatrix(int n, int profileSize, Matrix& result);
73
74 //-----
75 bool mul(const Matrix& a, const Matrix& b, Matrix& result);
76 bool sum(const Matrix& a, const Matrix& b, Matrix& result);
77
78 sumreal sumAllElementsAbs(const Matrix& a);
79
80 bool transpose(Matrix& a);
81
82 bool calcLDL(const Matrix& a, Matrix& l, Matrix& d);
83 bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x);
84 bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x);
85 bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x);
86 bool solveSLAE_by_LDL(const Matrix& a, const Matrix& y, Matrix& x);
87
88 bool solveSLAE_byGaussMethod(const Matrix& a, const Matrix& y, Matrix& x);

```

## 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,
67     ↪ because it's main diagonal
68     MatrixDiagonal(const Matrix& a);
69
70     void toDenseMatrix(Matrix& dense) const;
71     void resize(int n, std::vector<int> format); // format[0] must be 0, because
72     ↪ it's main diagonal
73
74     //-----
75     int dimension(void) const;
76     int getDiagonalsCount(void) const;
77     int getDiagonalSize(int diagNo) const;
78     int getDiagonalPos(int diagNo) const;
79
80     std::vector<int> getFormat(void) const;
81
82     //-----
83     matrix_diagonal_iterator posBegin(int diagNo) const;
84     matrix_diagonal_iterator posEnd(int diagNo) const;
85
86     iterator begin(int diagNo);
87     const_iterator begin(int diagNo) const;
88
89     iterator end(int diagNo);
90     const_iterator end(int diagNo) const;

```

```

90 private:
91     std::vector<std::vector<real>> di;
92     std::vector<int> fi;
93     int n;
94     Diagonal dc;
95 };
96
97 std::vector<int> makeSevenDiagonalFormat(int n, int m, int k);
98 std::vector<int> generateRandomFormat(int n, int diagonalsCount);
99
100 void generateDiagonalMatrix(
101     int n,
102     int min, int max,
103     std::vector<int> format,
104     MatrixDiagonal& result
105 );
106
107 void generateDiagonallyDominantMatrix(
108     int n,
109     std::vector<int> format,
110     bool isNegative,
111     MatrixDiagonal& result
112 );
113
114 bool mul(const MatrixDiagonal& a, const Vector& x, Vector& y);
115
116 //-----
117 /** Матричный "итератор" для движения по диагонали. */
118 class matrix_diagonal_iterator
119 {
120 public:
121     matrix_diagonal_iterator(int n, int d, bool isEnd);
122
123     matrix_diagonal_iterator& operator++();
124     matrix_diagonal_iterator operator++(int);
125
126     bool operator==(const matrix_diagonal_iterator& b) const;
127     bool operator!=(const matrix_diagonal_iterator& b) const;
128
129     matrix_diagonal_iterator& operator+=(const ptrdiff_t& movement);
130
131     int i, j;
132 };
133
134 /** Матричный "итератор" для движения по строке между различными диагоналями.
135     ↪ Может обрабатывать как всю строку, так и только нижний треугольник. */
136 class matrix_diagonal_line_iterator
137 {
138 public:
139     matrix_diagonal_line_iterator(int n, std::vector<int> format, bool
140         ↪ isOnlyLowTriangle);
141
142     matrix_diagonal_line_iterator& operator++();
143     matrix_diagonal_line_iterator operator++(int);
144
145     bool isLineEnd(void) const;
146     bool isEnd(void) const;
147
148     int i, j; // i - текущая строка, j - текущий столбец

```

```

147     int d, dn, di; // d - формат текущей диагонали, d - номер текущей диагонали,
    ↪ di - номер текущего элемента в диагонали
148 private:
149     std::map<int, int> m_map;
150     std::vector<int> m_sorted_format;
151
152     int line, start, end, pos;
153     Diagonal dc;
154
155     bool m_islineEnd;
156     bool m_isEnd;
157
158     void calcPos(void);
159 };
160
161 //-----
162 /** Класс итеративного решателя СЛАУ для диагональной матрицы. */
163 struct IterationsResult
164 {
165     int iterations;
166     double relativeResidual;
167 };
168
169 class SolverSLAE_Iterative
170 {
171 public:
172     SolverSLAE_Iterative();
173
174     IterationsResult jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x)
    ↪ const;
175     IterationsResult seidel(const MatrixDiagonal& a, const Vector& y, Vector& x)
    ↪ const;
176
177     double w;
178     bool isLog;
179     std::ostream& log;
180     Vector start;
181     double epsilon;
182     int maxIterations;
183
184 private:
185     mutable Vector x1;
186
187     // До итерации: x - текущее решение. После итерации x - следующее решение.
188     void iteration_jacobi(const MatrixDiagonal& a, const Vector& y, Vector& x)
    ↪ const;
189     void iteration_seidel(const MatrixDiagonal& a, const Vector& y, Vector& x)
    ↪ const;
190
191     typedef std::function<void(const SolverSLAE_Iterative*, const
    ↪ MatrixDiagonal&, const Vector&, Vector&)> step_function;
192
193     IterationsResult iteration_process(
194         const MatrixDiagonal& a,
195         const Vector& y,
196         Vector& x,
197         step_function step
198     ) const;
199 };

```

```

200
201 //-----
202 /** Класс итеративного решателя СЛАУ для плотной матрицы. */
203 class SolverSLAE_Iterative_matrix
204 {
205 public:
206     SolverSLAE_Iterative_matrix();
207
208     IterationsResult jacobi(const Matrix& a, const Vector& y, Vector& x) const;
209     IterationsResult seidel(const Matrix& a, const Vector& y, Vector& x) const;
210
211     double          w;
212     bool            isLog;
213     std::ostream&    log;
214     Vector          start;
215     double          epsilon;
216     int             maxIterations;
217 private:
218     mutable Vector x1;
219
220     // До итерации: x - текущее решение. После итерации x - следующее решение.
221     void iteration_jacobi(const Matrix& a, const Vector& y, Vector& x) const;
222     void iteration_seidel(const Matrix& a, const Vector& y, Vector& x) const;
223
224     typedef std::function<void(const SolverSLAE_Iterative_matrix*, const
225                               ↪ Matrix&, const Vector&, Vector&)> step_function;
226
227     IterationsResult iteration_process(
228         const Matrix& a,
229         const Vector& y,
230         Vector& x,
231         step_function step
232     ) const;
233 };

```

#### 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 << m_n << "\t" << m_m << std::endl;
31
32     fout.precision(std::numeric_limits<real>::digits10);
33     int w = std::numeric_limits<real>::digits10 + 4;
34
35     for (int i = 0; i < height(); ++i) {
36         for (int j = 0; j < width(); ++j)
37             fout << "\t" << operator()(i, j);
38         fout << std::endl;
39     }
40
41     fout.close();
42 }
43
44 //-----
45 void Matrix::getFromVector(int n, int m, const std::vector<real>& data) {
46     resize(n, m);
47     for (int i = 0; i < data.size(); ++i)
48         operator()(i / n, i % n) = data[i];
49 }
50
51 //-----
52 void Matrix::resize(int n, int m, real fill) {
53     if (m_n != n || m_m != m) {
54         m_n = n;

```

```

55     m_m = m;
56     m_matrix.clear();
57     m_matrix.resize(m_m, std::vector<real>(m_n, fill));
58 }
59 }
60
61 //-----
62 void Matrix::negate(void) {
63     for (auto& i : m_matrix) {
64         for (auto& j : i) {
65             j = -j;
66         }
67     }
68 }
69
70 //-----
71 bool Matrix::isSymmetric(void) const {
72     if (height() != width())
73         return false;
74
75     for (int i = 0; i < height(); ++i) {
76         for (int j = 0; j <= i; ++j) {
77             const real& a = operator()(i, j);
78             const real& b = operator()(j, i);
79             if (!isNear(a, b))
80                 return false;
81         }
82     }
83
84     return true;
85 }
86
87 //-----
88 bool Matrix::isLowerTriangular(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             if (fabs(operator()(j, i)) > 0.000001)
95                 return false;
96         }
97     }
98
99     return true;
100 }
101
102 //-----
103 bool Matrix::isUpperTriangular(void) const {
104     if (height() != width())
105         return false;
106
107     for (int i = 0; i < height(); ++i) {
108         for (int j = 0; j < i; ++j) {
109             if (operator()(i, j) != 0)
110                 return false;
111         }
112     }
113 }

```

```

114     return true;
115 }
116
117 //-----
118 bool Matrix::isDiagonal(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()(j, i) != 0 && operator()(i, j) != 0)
125                 return false;
126         }
127     }
128
129     return true;
130 }
131
132 //-----
133 bool Matrix::isDiagonalIdentity(void) const {
134     if (height() != width())
135         return false;
136
137     for (int i = 0; i < height(); ++i) {
138         if (operator()(i, i) != 1)
139             return false;
140     }
141
142     return true;
143 }
144
145 //-----
146 bool Matrix::isDegenerate(void) const {
147     // TODO
148     return false;
149 }
150
151 //-----
152 real& Matrix::operator()(int i, int j) {
153     return m_matrix[i][j];
154 }
155
156 //-----
157 const real& Matrix::operator()(int i, int j) const {
158     return m_matrix[i][j];
159 }
160
161 //-----
162 int Matrix::width(void) const {
163     return m_n;
164 }
165
166 //-----
167 int Matrix::height(void) const {
168     return m_m;
169 }
170
171 //-----
172 //-----

```

```

173 //-----
174
175 //-----
176 void generateSparseSymmetricMatrix(int n, int min, int max, real percent,
    ↪ Matrix& result) {
177     result.resize(n, n, 0);
178
179     int count = percent * n * n;
180
181     for (int k = 0; k < count; ++k) {
182         int i = intRandom(0, n-1);
183         int j = intRandom(0, i-1);
184         result(i, j) = intRandom(min, max);
185         result(j, i) = result(i, j);
186     }
187
188     for (int i = 0; i < n; i++)
189         result(i, i) = intRandom(1, max - min);
190 }
191
192 //-----
193 void generateLMatrix(int n, int min, int max, real percent, Matrix& result) {
194     result.resize(n, n, 0);
195
196     int count = percent * n * n / 2;
197
198     for (int k = 0; k < count; ++k) {
199         int i = intRandom(0, n-1);
200         int j = intRandom(0, i-1);
201         result(i, j) = intRandom(min, max);
202     }
203
204     for (int i = 0; i < n; ++i) {
205         result(i, i) = 1;
206     }
207 }
208
209 //-----
210 void generateDiagonalMatrix(int n, int min, int max, Matrix& result) {
211     result.resize(n, n, 0);
212
213     for (int i = 0; i < n; ++i)
214         result(i, i) = intRandom(min, max);
215 }
216
217 //-----
218 void generateVector(int n, int min, int max, Matrix& result) {
219     result.resize(1, n, 0);
220
221     for (int i = 0; i < n; ++i)
222         result(i, 0) = intRandom(min, max);
223 }
224
225 //-----
226 void generateVector(int n, Matrix& result) {
227     result.resize(1, n, 0);
228
229     for (int i = 0; i < n; ++i)
230         result(i, 0) = i+1;

```

```

231 }
232
233 //-----
234 void generateGilbertMatrix(int n, Matrix& result) {
235     result.resize(n, n);
236
237     for (int i = 0; i < n; ++i) {
238         for (int j = 0; j < n; ++j) {
239             result(i, j) = double(1.0)/double((i+1)+(j+1)-1);
240         }
241     }
242 }
243
244 //-----
245 void generateTestMatrix(int n, int profileSize, Matrix& result) {
246     result.resize(n, n);
247
248     for (int i = 0; i < n; ++i) {
249         for (int j = 0; j < profileSize; ++j) if (i-j-1 >= 0) {
250             result(i, i-j-1) = -intRandom(0, 5);
251             result(i-j-1, i) = result(i, i-j-1);
252         }
253     }
254
255     for (int i = 0; i < n; ++i) {
256         sumreal sum = 0;
257         for (int j = 0; j < n; ++j) if (i != j) {
258             sum += result(i, j);
259         }
260         result(i, i) = -sum;
261     }
262 }
263
264 //-----
265 //-----
266 //-----
267
268 //-----
269 bool mul(const Matrix& a, const Matrix& b, Matrix& result) {
270     // result = rus_a * b
271     if (a.width() != b.height())
272         return false;
273
274     result.resize(b.width(), a.height());
275
276     for (int i = 0; i < b.width(); ++i) {
277         for (int j = 0; j < a.height(); ++j) {
278             real sum = 0;
279             for (int k = 0; k < a.width(); ++k) {
280                 sum += a(j, k) * b(k, i);
281             }
282             result(j, i) = sum;
283         }
284     }
285
286     return true;
287 }
288
289 //-----

```

```

290 bool sum(const Matrix& a, const Matrix& b, Matrix& result) {
291     // result = rus_a + b
292     if (a.width() != b.width() || a.height() != b.height())
293         return false;
294
295     result.resize(a.width(), a.height());
296
297     for (int i = 0; i < a.width(); ++i) {
298         for (int j = 0; j < a.height(); ++j) {
299             result(j, i) = a(j, i) + b(j, i);
300         }
301     }
302
303     return true;
304 }
305
306 //-----
307 bool transpose(Matrix& a) {
308     // rus_a = rus_a^T
309     if (a.height() != a.width())
310         return false;
311
312     for (int i = 0; i < a.height(); ++i) {
313         for (int j = 0; j < i; ++j) {
314             std::swap(a(j, i), a(i, j));
315         }
316     }
317
318     return true;
319 }
320
321 //-----
322 sumreal sumAllElementsAbs(const Matrix& a) {
323     sumreal sum = 0;
324     for (int i = 0; i < a.height(); ++i) {
325         for (int j = 0; j < a.width(); ++j) {
326             sum += fabs(a(i, j));
327         }
328     }
329
330     return sum;
331 }
332
333 //-----
334 bool calcLDL(const Matrix& a, Matrix& l, Matrix& d) {
335     // l * d * l^T = rus_a
336     if (!a.isSymmetric())
337         return false;
338
339     l.resize(a.width(), a.height(), 0);
340     d.resize(a.width(), a.height(), 0);
341
342     for (int i = 0; i < a.height(); ++i) {
343         // Считаем элементы матрицы L
344         for (int j = 0; j < i; ++j) {
345             real sum = 0;
346             for (int k = 0; k < j; ++k)
347                 sum += d(k, k) * l(j, k) * l(i, k);
348

```

```

349         if (fabs(d(j, j)) < 0.0001)
350             l(i, j) = 0;
351         else
352             l(i, j) = (a(i, j) - sum) / d(j, j);
353     }
354
355     // Считаем диагональный элемент
356     {
357         real sum = 0;
358         for (int j = 0; j < i; ++j)
359             sum += d(j, j) * l(i, j) * l(i, j);
360         d(i, i) = a(i, i) - sum;
361     }
362 }
363
364 for (int i = 0; i < l.height(); i++)
365     l(i, i) = 1;
366
367 return true;
368 }
369
370 //-----
371 bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x) {
372     // l * x = y, l - нижнетреугольная матрица
373     if (!l.isLowerTriangular() || !l.isDiagonalIdentity())
374         return false;
375
376     x.resize(1, y.height());
377
378     for (int i = x.height() - 1; i >= 0; --i) {
379         real sum = 0;
380         for (int j = i; j < x.height(); ++j)
381             sum += l(j, i) * x(j, 0);
382         x(i, 0) = y(i, 0) - sum;
383     }
384
385     return true;
386 }
387
388 //-----
389 bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x) {
390     // l * x = y, l - верхнетреугольная матрица
391     if (!l.isLowerTriangular() || !l.isDiagonalIdentity())
392         return false;
393
394     x.resize(1, y.height());
395
396     for (int i = 0; i < x.height(); ++i) {
397         real sum = 0;
398         for (int j = 0; j < i; ++j)
399             sum += l(i, j) * x(j, 0);
400         x(i, 0) = y(i, 0) - sum;
401     }
402
403     return true;
404 }
405
406 //-----
407 bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x) {

```

```

408 // d * x = y, d - диагональная матрица
409 if (!d.isDiagonal())
410     return false;
411
412 x.resize(1, y.height());
413
414 for (int i = 0; i < x.height(); ++i)
415     x(i, 0) = y(i, 0) / d(i, i);
416
417 return true;
418 }
419
420 //-----
421 bool solveSLAE_by_LDL(const Matrix& a, const Matrix& y, Matrix& x) {
422     // rus_a * x = y, rus_a - симметричная матрица
423     if (!(a.width() == a.height() && a.width() == y.height() &&
424         ↪ !a.isDegenerate()))
425         return false;
426
427     Matrix l, d, z, w;
428
429     if (!calcLDL(a, l, d))
430         return false;
431
432     if (!calcGaussianFrontOrder(l, y, z))
433         return false;
434
435     if (!calcGaussianCentralOrder(d, z, w))
436         return false;
437
438     if (!calcGaussianReverseOrder(l, w, x))
439         return false;
440
441     return true;
442 }
443 //-----
444 bool solveSLAE_byGaussMethod(const Matrix& a1, const Matrix& y1, Matrix& x1) {
445     if (!(a1.width() == a1.height() && y1.height() == a1.width() &&
446         ↪ !a1.isDegenerate()))
447         return false;
448
449     Matrix a(a1);
450     Matrix y(y1);
451
452     for (int i = 0; i < a.height(); ++i) {
453         // Находим максимальный элемент
454         int maxI = i;
455         for (int j = i+1; j < a.height(); ++j)
456             if (fabs(a(j, i)) > fabs(a(maxI, i)))
457                 maxI = j;
458
459         // Переставляем эту строчку с текущей
460         for (int j = i; j < a.width(); ++j)
461             std::swap(a(i, j), a(maxI, j));
462         std::swap(y(i, 0), y(maxI, 0));
463
464         // Перебираем все строчки ниже и отнимаем текущую строчку от них
465         for (int j = i+1; j < a.height(); ++j) {

```



```

465     real m = a(j, i) / a(i, i);
466     for (int k = i; k < a.width(); ++k)
467         a(j, k) -= m * a(i, k);
468     y(j, 0) -= m * y(i, 0);
469 }
470
471 // Делим текущую строку на ее ведущий элемент, чтобы на диагонали были
472   ↳ единицы
473 double m = a(i, i);
474 for (int j = i; j < a.width(); ++j)
475     a(i, j) /= m;
476 y(i, 0) /= m;
477 }
478 // Считаем обратный ход Гаусса
479 transpose(a);
480 calcGaussianReverseOrder(a, y, x1);
481
482 return true;
483 }

```

### 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, 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 int MatrixDiagonal::dimension(void) const {
173     return n;
174 }
175
176 //-----
177 int MatrixDiagonal::getDiagonalsCount(void) const {
178     return di.size();
179 }
180
181 //-----
182 int MatrixDiagonal::getDiagonalSize(int diagNo) const {
183     return di[diagNo].size();
184 }
185
186 //-----
187 int MatrixDiagonal::getDiagonalPos(int diagNo) const {
188     return fi[diagNo];
189 }
190
191 //-----
192 std::vector<int> MatrixDiagonal::getFormat(void) const {
193     return fi;
194 }
195
196 //-----
197 matrix_diagonal_iterator MatrixDiagonal::posBegin(int diagNo) const {
198     return matrix_diagonal_iterator(n, fi[diagNo], false);
199 }
200
201 //-----
202 matrix_diagonal_iterator MatrixDiagonal::posEnd(int diagNo) const {
203     return matrix_diagonal_iterator(n, fi[diagNo], true);
204 }
205
206 //-----
207 MatrixDiagonal::iterator MatrixDiagonal::begin(int diagNo) {
208     return di[diagNo].begin();
209 }
210
211 //-----
212 MatrixDiagonal::const_iterator MatrixDiagonal::begin(int diagNo) const {
213     return di[diagNo].begin();
214 }

```

```

215
216 //-----
217 MatrixDiagonal::iterator MatrixDiagonal::end(int diagNo) {
218     return di[diagNo].end();
219 }
220
221 //-----
222 MatrixDiagonal::const_iterator MatrixDiagonal::end(int diagNo) const {
223     return di[diagNo].end();
224 }
225
226 //-----
227 //-----
228 //-----
229
230 //-----
231 matrix_diagonal_iterator::matrix_diagonal_iterator(int n, int d, bool isEnd) {
232     Diagonal dc(n);
233     if (isEnd) {
234         i = dc.calcLine_byDP(d, dc.calcDiagonalSize(d));
235         j = dc.calcRow_byDP(d, dc.calcDiagonalSize(d));
236     } else {
237         i = dc.calcLine_byDP(d, 0);
238         j = dc.calcRow_byDP(d, 0);
239     }
240 }
241
242 //-----
243 matrix_diagonal_iterator& matrix_diagonal_iterator::operator++() {
244     i++;
245     j++;
246     return *this;
247 }
248
249 //-----
250 matrix_diagonal_iterator matrix_diagonal_iterator::operator++(int) {
251     i++;
252     j++;
253     return *this;
254 }
255
256 //-----
257 bool matrix_diagonal_iterator::operator==(const matrix_diagonal_iterator& b)
258     ⇨ const {
259     return b.i == i && b.j == j;
260 }
261
262 //-----
263 bool matrix_diagonal_iterator::operator!=(const matrix_diagonal_iterator& b)
264     ⇨ const {
265     return b.i != i || b.j != j;
266 }
267
268 //-----
269 matrix_diagonal_iterator& matrix_diagonal_iterator::operator+=(const ptrdiff_t&
270     ⇨ movement) {
271     i += movement;
272     j += movement;
273     return *this;

```

```

271 }
272
273 //-----
274 //-----
275 //-----
276
277 //-----
278 matrix_diagonal_line_iterator::matrix_diagonal_line_iterator(int n,
↪ std::vector<int> format, bool isOnlyLowTriangle) : dc(n), m_isEnd(false),
↪ m_isLineEnd(false) {
279     // Создаем обратное преобразование из формата диагонали в ее номер в формате
280     for (int i = 0; i < format.size(); ++i)
281         if ((isOnlyLowTriangle && format[i] < 0) || !isOnlyLowTriangle)
282             m_map[format[i]] = i;
283
284     // Создаем сортированный формат, чтобы по нему двигаться
285     if (isOnlyLowTriangle) {
286         for (int i = 0; i < format.size(); ++i)
287             if (format[i] < 0)
288                 m_sorted_format.push_back(format[i]);
289     } else
290         m_sorted_format = format;
291     std::sort(m_sorted_format.begin(), m_sorted_format.end());
292
293     line = 0;
294     pos = 0;
295     start = m_sorted_format.size() - 1;
296     end = start;
297
298     // Находим, с какой диагонали начинается текущая строка
299     for (int i = 0; i < m_sorted_format.size(); ++i) {
300         if (dc.isLineIntersectDiagonal(line, m_sorted_format[i])) {
301             start = i;
302             break;
303         }
304     }
305
306     // Находим на какой диагонали кончается текущая строка
307     for (int i = 0; i < m_sorted_format.size(); ++i) {
308         int j = m_sorted_format.size() - i - 1;
309         if (dc.isLineIntersectDiagonal(line, m_sorted_format[j])) {
310             end = j;
311             break;
312         }
313     }
314
315     calcPos();
316 }
317
318 //-----
319 matrix_diagonal_line_iterator& matrix_diagonal_line_iterator::operator++() {
320     if (!m_isEnd) {
321         if (m_isLineEnd) {
322             // Сдвигаемся на одну строку
323             line++;
324
325             // Определяем какие диагонали пересекают эту строку
326             if (start != 0)
327                 if (dc.isLineIntersectDiagonal(line, m_sorted_format[start-1]))

```

```

328         start = start-1;
329
330         if (end != 0)
331             if (!dc.isLineIntersectDiagonal(line, m_sorted_format[end]))
332                 if (start != end)
333                     end = end-1;
334
335         m_isLineEnd = false;
336         if (line == dc.n)
337             m_isEnd = true;
338
339         pos = 0;
340         calcPos();
341     } else {
342         // Сдвигаемся на один столбец
343         pos++;
344         calcPos();
345     }
346 }
347
348 return *this;
349 }
350
351 //-----
352 matrix_diagonal_line_iterator matrix_diagonal_line_iterator::operator++(int) {
353     return operator++();
354 }
355
356 //-----
357 bool matrix_diagonal_line_iterator::isLineEnd(void) const {
358     return m_isLineEnd;
359 }
360
361 //-----
362 bool matrix_diagonal_line_iterator::isEnd(void) const {
363     return m_isEnd;
364 }
365
366 //-----
367 void matrix_diagonal_line_iterator::calcPos(void) {
368     // Вычисляет все текущие положения согласно переменным start, pos и формату
369     if (!dc.isLineIntersectDiagonal(line, m_sorted_format[end]) || (start + pos
370     ↪ > end)) {
371         m_isLineEnd = true;
372         i = line;
373         j = 0;
374         d = 0;
375         di = 0;
376         dn = 0;
377     } else {
378         i = line;
379         d = m_sorted_format[start + pos];
380         dn = m_map[d];
381         di = dc.calcPos_byDL(d, i);
382         j = dc.calcRow_byDL(d, i);
383     }
384 }
385 //-----

```

```

386 //-----
387 //-----
388
389 //-----
390 std::vector<int> makeSevenDiagonalFormat(int n, int m, int k) {
391     std::vector<int> result;
392
393     if (1+m+k >= n)
394         throw std::exception();
395
396     result.push_back(0);
397
398     result.push_back(1);
399     result.push_back(1+m);
400     result.push_back(1+m+k);
401
402     result.push_back(-1);
403     result.push_back(-1-m);
404     result.push_back(-1-m-k);
405
406     return result;
407 }
408
409 //-----
410 std::vector<int> generateRandomFormat(int n, int diagonalsCount) {
411     Diagonal d(n);
412
413     std::vector<int> result;
414     result.push_back(0);
415
416     // Создаем массив всех возможных диагоналей
417     std::vector<int> diagonals;
418     for (int i = d.calcMinDiagonal(); i <= d.calcMaxDiagonal(); ++i)
419         if (i != 0)
420             diagonals.push_back(i);
421
422     diagonalsCount = std::min<int>(diagonals.size(), diagonalsCount);
423
424     // Заполняем результат случайными диагоналями из этого массива
425     for (int i = 0; i < diagonalsCount; ++i) {
426         int pos = intRandom(0, diagonals.size());
427         result.push_back(diagonals[pos]);
428         diagonals.erase(diagonals.begin() + pos);
429     }
430
431     return result;
432 }
433
434 //-----
435 void generateDiagonalMatrix(int n, int min, int max, std::vector<int> format,
436     ↪ MatrixDiagonal& result) {
437     result.resize(n, format);
438     for (int i = 0; i < result.getDiagonalsCount(); ++i) {
439         auto mit = result.posBegin(i);
440         for (auto it = result.begin(i); it != result.end(i); ++it, ++mit)
441             (*it) = intRandom(min, max);
442     }
443 }

```



```

444 //-----
445 void generateDiagonallyDominantMatrix(int n, std::vector<int> format, bool
↪ isNegative, MatrixDiagonal& result) {
446     result.resize(n, format);
447
448     for (int i = 0; i < result.getDiagonalsCount(); ++i) {
449         auto mit = result.posBegin(i);
450         for (auto it = result.begin(i); it != result.end(i); ++it, ++mit) {
451             if (isNegative)
452                 *it = -intRandom(0, 5);
453             else
454                 *it = intRandom(0, 5);
455         }
456     }
457
458     matrix_diagonal_line_iterator mit(n, format, false);
459     for (; !mit.isEnd(); ++mit) {
460         sumreal& sum = result.begin(0)[mit.i];
461         sum = 0;
462         for (; !mit.isLineEnd(); ++mit)
463             if (mit.i != mit.j)
464                 sum += result.begin(mit.dn)[mit.di];
465         sum = std::fabs(sum);
466     }
467
468     result.begin(0)[0] += 1;
469 }
470
471 //-----
472 bool mul(const MatrixDiagonal& a, const Vector& x, Vector& y) {
473     if (x.size() != a.dimension())
474         return false;
475
476     y.resize(x.size());
477
478     // Зануление результата
479     y.zero();
480
481     for (int i = 0; i < a.getDiagonalsCount(); ++i) {
482         auto mit = a.posBegin(i);
483         for (auto it = a.begin(i); it != a.end(i); ++it, ++mit)
484             y(mit.i) += (*it) * x(mit.j);
485     }
486
487     return true;
488 }
489
490 //-----
491 //-----
492 //-----
493
494 //-----
495 SolverSLAE_Iterative::SolverSLAE_Iterative() :
496     w(1),
497     isLog(false),
498     log(std::cout),
499     start(),
500     epsilon(0.00001),
501     maxIterations(100) {

```

```

502 }
503
504 //-----
505 IterationsResult SolverSLAE_Iterative::jacobi(const MatrixDiagonal& a, const
↪ Vector& y, Vector& x) const {
506     return iteration_process(a, y, x, &SolverSLAE_Iterative::iteration_jacobi);
507 }
508
509 //-----
510 IterationsResult SolverSLAE_Iterative::seidel(const MatrixDiagonal& a, const
↪ Vector& y, Vector& x) const {
511     return iteration_process(a, y, x, &SolverSLAE_Iterative::iteration_seidel);
512 }
513
514 //-----
515 IterationsResult SolverSLAE_Iterative::iteration_process(const MatrixDiagonal&
↪ a, const Vector& y, Vector& x, step_function step) const {
516     if (a.dimension() != y.size() || start.size() != y.size())
517         throw std::exception();
518
519     // Считаем норму матрицы: ее максимальный элемент по модулю
520     real yNorm = calcNorm(y);
521     x1.resize(y.size());
522     x = start;
523
524     // Цикл по итерациям
525     int i = 0;
526     real relativeResidual = epsilon + 1;
527     for (; i < maxIterations && relativeResidual > epsilon; ++i) {
528         // Итерационный шаг
529         step(this, a, y, x);
530
531         // Считаем невязку
532         mul(a, x, x1);
533         x1.negate();
534         sum(x1, y, x1);
535         relativeResidual = fabs(calcNorm(x1)) / yNorm;
536
537         // Выводим данные
538         if (isLog)
539             log << i << "\t" << std::scientific << std::setprecision(3) <<
↪ relativeResidual << std::endl;
540     }
541
542     return {i, relativeResidual};
543 }
544
545 //-----
546 void SolverSLAE_Iterative::iteration_jacobi(const MatrixDiagonal& a, const
↪ Vector& y, Vector& x) const {
547     // Умножаем матрицу на решение
548     mul(a, x, x1);
549
550     //  $x^{(k+1)} = x^{(k)} + w/a(i, i) * x^{(k+1)}$ 
551     auto it = a.begin(0);
552     for (int i = 0; i < x1.size(); ++i, ++it)
553         x(i) += w / (*it) * (y(i) - x1(i));
554 }
555

```

```

556 //-----
557 void SolverSLAE_Iterative::iteration_seidel(const MatrixDiagonal& a, const
↪ Vector& y, Vector& x) const {
558     // Умножаем верхний треугольник на решение
559     x1.zero();
560     for (int i = 0; i < a.getDiagonalsCount(); ++i)
561         if (a.getDiagonalPos(i) >= 0) {
562             auto mit = a.posBegin(i);
563             for (auto it = a.begin(i); it != a.end(i); ++it, ++mit)
564                 x1(mit.i) += (*it) * x(mit.j);
565         }
566
567     // Проходим по нижнему треугольнику и считаем все параметры
568     matrix_diagonal_line_iterator mit(a.dimension(), a.getFormat(), true);
569     for (; !mit.isEnd(); ++mit) {
570         for (; !mit.isLineEnd(); ++mit)
571             x1(mit.i) += a.begin(mit.dn)[mit.di] * x(mit.j);
572         x(mit.i) = x(mit.i) + w/a.begin(0)[mit.i] * (y(mit.i) - x1(mit.i));
573     }
574 }
575
576 //-----
577 //-----
578 //-----
579
580 //-----
581 SolverSLAE_Iterative_matrix::SolverSLAE_Iterative_matrix() :
582     w(1),
583     isLog(false),
584     log(std::cout),
585     start(),
586     epsilon(0.00001),
587     maxIterations(100) {
588 }
589
590 //-----
591 IterationsResult SolverSLAE_Iterative_matrix::jacobi(const Matrix& a, const
↪ Vector& y, Vector& x) const {
592     return iteration_process(a, y, x,
↪ &SolverSLAE_Iterative_matrix::iteration_jacobi);
593 }
594
595 //-----
596 IterationsResult SolverSLAE_Iterative_matrix::seidel(const Matrix& a, const
↪ Vector& y, Vector& x) const {
597     return iteration_process(a, y, x,
↪ &SolverSLAE_Iterative_matrix::iteration_seidel);
598 }
599
600 //-----
601 void SolverSLAE_Iterative_matrix::iteration_jacobi(const Matrix& a, const
↪ Vector& y, Vector& x) const {
602     for (int i = 0; i < a.height(); ++i) {
603         sumreal sum = 0;
604         for (int j = 0; j < a.height(); ++j)
605             sum += a(i, j) * x(j);
606         x1(i) = x(i) + w / a(i, i) * (y(i) - sum);
607     }
608 }

```

```

609     x = x1;
610 }
611
612 //-----
613 void SolverSLAE_Iterative_matrix::iteration_seidel(const Matrix& a, const
↪ Vector& y, Vector& x) const {
614     for (int i = 0; i < a.height(); ++i) {
615         sumreal sum = 0;
616         for (int j = 0; j < a.height(); ++j)
617             sum += a(i, j) * x(j);
618         x(i) = x(i) + w / a(i, i) * (y(i) - sum);
619     }
620 }
621
622 //-----
623 IterationsResult SolverSLAE_Iterative_matrix::iteration_process(const Matrix& a,
↪ const Vector& y, Vector& x, step_function step) const {
624     if (a.width() != a.height() || a.width() != y.size() || start.size() !=
↪ y.size())
625         throw std::exception();
626
627     // Считаем норму матрицы: ее максимальный элемент по модулю
628     real yNorm = calcNorm(y);
629     x1.resize(y.size());
630     x = start;
631
632     // Цикл по итерациям
633     int i = 0;
634     real relativeResidual = epsilon + 1;
635     for (; i < maxIterations && relativeResidual > epsilon; ++i) {
636         // Итерационный шаг
637         step(this, a, y, x);
638
639         // Считаем невязку
640         mul(a, x, x1);
641         x1.negate();
642         sum(x1, y, x1);
643         relativeResidual = calcNorm(x1) / yNorm;
644
645         // Выводим данные
646         if (isLog)
647             log << i << "\t" << std::scientific << std::setprecision(3) <<
↪ relativeResidual << std::endl;
648     }
649
650     return {i, relativeResidual};
651 }

```

#### FILE table\_generator.cpp

```

1 #include <fstream>
2 #include <cmath>
3 #include <iomanip>
4 #include <algorithm>
5 #include "diagonal.h"
6
7 typedef std::function<IterationsResult(const SolverSLAE_Iterative*, const
↪ MatrixDiagonal& a, const Vector& y, Vector& x)> method_function;
8
9 //-----

```

```

10 void makeTable(
11     const MatrixDiagonal& a,
12     const Vector& x_precise,
13     const Vector& y,
14     SolverSLAE_Iterative& solver,
15     std::string fileName
16 ) {
17     std::ofstream fout(fileName + ".tex");
18     std::ofstream fout1(fileName + ".dat");
19
20     // Выводим матрицу и остальное в виде формулы
21     auto format = a.getFormat();
22     fout << "$$ " << fileName << "=\left(\quad\begin{matrix}\n";
23     Matrix a_dense;
24     a.toDenseMatrix(a_dense);
25     for (int i = 0; i < a_dense.height(); i++) {
26         for (int j = 0; j < a_dense.width(); j++) {
27             int d = Diagonal(a_dense.height()).calcDiag_byLR(i, j);
28             bool isOnFormat = std::find(format.begin(), format.end(), d) !=
29                 format.end();
30             if (isOnFormat)
31                 fout << "\\cellcolor{green!30}";
32             fout << int(a_dense(i, j));
33             if (j + 1 != a_dense.width())
34                 fout << " & ";
35         }
36         if (i + 1 != a_dense.height())
37             fout << " \\n";
38         else
39             fout << " \n";
40     }
41     fout << "\\end{matrix}\\quad\\right), X=\\begin{pmatrix}";
42     for (int i = 0; i < x_precise.size(); i++) {
43         if (i + 1 != x_precise.size())
44             fout << int(x_precise(i)) << " \\n";
45         else
46             fout << int(x_precise(i)) << " \n";
47     }
48     fout << "\\end{pmatrix}, F=\\begin{pmatrix}";
49     for (int i = 0; i < y.size(); i++) {
50         if (i + 1 != y.size())
51             fout << int(y(i)) << " \\n";
52         else
53             fout << int(y(i)) << " \n";
54     }
55     fout << "\\end{pmatrix} $$\n\n";
56
57     // Выводим параметры решателя
58     int exponent = floor(log10(solver.epsilon));
59     double number = solver.epsilon / pow(10.0, exponent);
60     fout
61         << "$$ \\varepsilon = ";
62     if (fabs(number - 1) >= 0.01)
63         fout
64             << std::setprecision(2) << std::fixed << number
65             << " \\cdot ";
66     fout
67         << "10^{ " << exponent << " }, \\quad iterations_{max} = "
68         << solver.maxIterations << ", \\quad start = \\begin{pmatrix} ";

```

```

68 fout << std::defaultfloat;
69 for (int i = 0; i < solver.start.size(); i++) {
70     fout << solver.start(i);
71     if (i + 1 != solver.start.size())
72         fout << " & ";
73     else
74         fout << " ";
75 }
76 fout << "\\end{pmatrix}^T $$\\n\\n";
77
78 std::vector<double> w1(200), w2(200);
79 std::vector<Vector> x1(200), x2(200);
80 std::vector<Vector> xsub1(200), xsub2(200);
81 std::vector<double> rr1(200), rr2(200); // relativeResidual
82 std::vector<double> va1(200), va2(200);
83 std::vector<int> it1(200), it2(200);
84
85 int min1, min2;
86 int count1, count2;
87
88 auto one_method = [&a, &x_precise, &y, &solver] (
89     std::vector<double>& w,
90     std::vector<Vector>& x,
91     std::vector<Vector>& xsub,
92     std::vector<double>& rr,
93     std::vector<double>& va,
94     std::vector<int>& it,
95
96     int& min,
97     int& count,
98
99     method_function method
100 ) {
101     min = 0;
102     count = 200;
103     Vector x_solve(x_precise.size());
104     Vector x_sub(x_precise.size());
105     real xNorm = calcNorm(x_precise);
106
107     for (int i = 0; i < 200; ++i) {
108         solver.w = i / 100.0;
109         auto result = method(&solver, a, y, x_solve);
110
111         // Если начинается ошибки после 100 итерации, то и потом ничего
112         ↪ кроме них не будет, поэтому заканчиваем цикл
113         if ((result.relativeResidual > solver.epsilon && i >= 100) ||
114             (result.relativeResidual != result.relativeResidual)) {
115             count = i;
116             break;
117         }
118
119         // Вычисления разности точного и приближенного решений
120         x_sub = x_solve;
121         x_sub.negate();
122         sum(x_sub, x_precise, x_sub);
123         real x_subNorm = calcNorm(x_sub);
124
125         w[i] = solver.w;
126         x[i] = x_solve;

```

```

126     xsub[i] = x_sub;
127     rr[i] = result.relativeResidual;
128     va[i] = x_subNorm / xNorm / result.relativeResidual;
129     it[i] = result.iterations;
130
131     // Находим минимум
132     if (result.iterations < it[min])
133         min = i;
134 }
135 };
136
137 one_method(w1, x1, xsub1, rr1, va1, it1, min1, count1,
138     ↪ &SolverSLAE_Iterative::jacobi);
139 one_method(w2, x2, xsub2, rr2, va2, it2, min2, count2,
140     ↪ &SolverSLAE_Iterative::seidel);
141
142 // w, x, x-x*, относительная невязка, vA, итераций
143 fout
144 << "\\setlength{\\tabcolsep}{2pt}\\n"
145 << "\\tabulinesep=0.3mm\\n"
146 << "\\noindent{\\scriptsize\\texttt{"
147 << "\\begin{longtabu}{\\n"
148 << "|X[-1,c]|X[-1,c]|X[-1,c]|X[-1,c]|X[-1,c]|X[-1,c]|\\n"
149 << "p{0.05cm}\\n|X[-1,c]|X[-1,c]|X[-1,c]|X[-1,c]|X[-1,c]|X[-1,c]|}\\n"
150 << "\\cline{1-6}\\cline{8-13}\\n"
151 << "\\multicolumn{6}{|c|}{Метод Якоби} && \\multicolumn{6}{|c|}{Метод
152     ↪ Зейделя} \\\\n"
153 << "\\cline{1-6}\\cline{8-13}\\n";
154
155 auto write_vector = [&fout] (const Vector& a) {
156     fout << "\\tcell{";
157     for (int i = 0; i < a.size(); ++i)
158         if (i + 1 != a.size())
159             fout << a[i] << " \\\\ ";
160         else
161             fout << a[i] << "}";
162 };
163
164 auto write_line = [&] (int i, int colorNo) {
165     int doublePrec = 16;
166     if (i < count1) {
167         if (colorNo == 1) {
168             fout << std::fixed << std::setprecision(2) <<
169                 ↪ "\\cellcolor{green!30}" << w1[i] << "} & ";
170             fout << std::fixed << std::setprecision(doublePrec) <<
171                 ↪ "\\tiny{\\cellcolor{green!30}}";
172             write_vector(x1[i]);
173             fout << "}" & " << "\\tiny{\\cellcolor{green!30}}";
174             fout << std::scientific << std::setprecision(1);
175             write_vector(xsub1[i]);
176             fout << "}" & " << std::scientific << std::setprecision(1) <<
177                 ↪ "\\cellcolor{green!30}" << rr1[i] << "} & ";
178             fout << std::fixed << std::setprecision(2) <<
179                 ↪ "\\cellcolor{green!30}" << va1[i] << "} & ";
180             fout << "\\cellcolor{green!30}" << it1[i] << "} & ";
181         } else {
182             fout << std::fixed << std::setprecision(2) << w1[i] << " & ";
183             fout << std::fixed << std::setprecision(doublePrec) << "\\tiny{";

```





```

231         } else {
232             write_line(i, 0);
233         }
234     }
235     if (i + 10 > min1 && min1 > i) {
236         write_line(min1, 1);
237     } else {
238         if (i + 10 > min2 && min2 > i) {
239             write_line(min2, 2);
240         }
241     }
242 }
243
244 fout1 << "w1\tit1\tw2\tit2" << std::endl;
245 fout1 << std::fixed << std::setprecision(2);
246 for (int i = 0; i < std::max(count1, count2); ++i) {
247     if (i >= count1)
248         fout1 << w1[count1-1] << "\t" << it1[count1-1] << "\t";
249     else
250         fout1 << w1[i] << "\t" << it1[i] << "\t";
251     if (i >= count2)
252         fout1 << w2[count2-1] << "\t" << it2[count2-1] << std::endl;
253     else
254         fout1 << w2[i] << "\t" << it2[i] << std::endl;
255 }
256
257 fout << "\\end{longtabu}}}\n\n";
258
259 fout
260 << "\\noindent\\begin{tikzpicture}\n"
261 << "\\begin{semilogyaxis}[xlabel=w,ylabel=Iterations,width=\\textwidth,"
262   << "height=6cm]\n"
263 << "\\addplot[red, no markers] table [y=it1, x=w1]{\n" << fileName <<
264   << ".dat};\n"
265 << "\\addplot[blue, no markers] table [y=it2, x=w2]{\n" << fileName <<
266   << ".dat};\n"
267 << "\\legend{Jacobi,Seidel}\n"
268 << "\\end{semilogyaxis}\n"
269 << "\\end{tikzpicture}";
270
271 fout.close();
272 fout1.close();
273 }
274
275
276 //-----
277 //-----
278 //-----
279
280 int main() {
281     // Получаем необходимые матрицы
282     MatrixDiagonal a, b;
283     generateDiagonallyDominantMatrix(10, makeSevenDiagonalFormat(10, 3, 2),
284   << true, a);
285     generateDiagonallyDominantMatrix(10, makeSevenDiagonalFormat(10, 2, 4),
286   << false, b);
287
288     Vector x;
289     x.generate(10);
290

```

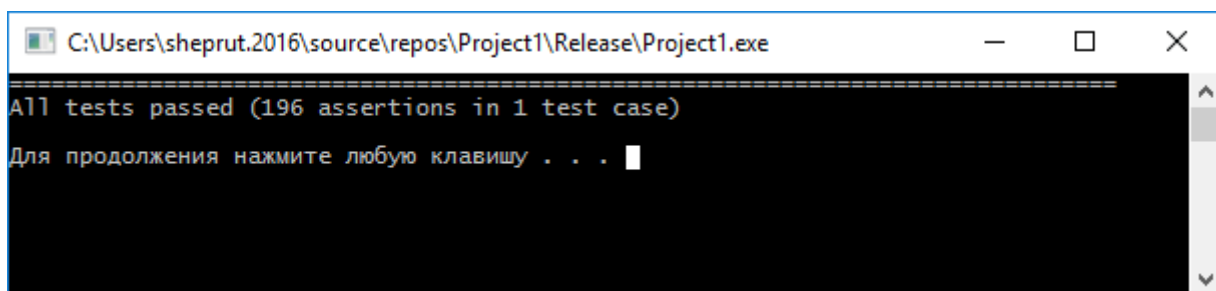
```

285 Vector y_a, y_b;
286 mul(a, x, y_a);
287 mul(b, x, y_b);
288
289 // Начальные присвоения
290 SolverSLAE_Iterative solver;
291 solver.w = 0;
292 solver.isLog = false;
293 solver.start = Vector(10, 0);
294 solver.epsilon = 1e-14;
295 solver.maxIterations = 1e5;
296
297 // Создаем таблицы
298 makeTable(a, x, y_a, solver, "A");
299 makeTable(b, x, y_b, solver, "B");
300 }

```

### 3 Тестирование

Для тестирования использовалось юнит-тестирование и библиотека Catch. Было протестировано получение необходимой относительной невязки на матрицах с диагональным преобладанием.



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\sheprut.2016\source\repos\Project1\Release\Project1.exe'. The window content displays the text 'All tests passed (196 assertions in 1 test case)' in green, followed by 'Для продолжения нажмите любую клавишу . . .' in white on a black background. A cursor is visible at the end of the second line.

#### FILE diagonal\_test.cpp

```

1 #define CATCH_CONFIG_RUNNER
2
3 #include "../1/catch.hpp"
4 #include "../1/matrix.h"
5 #include "../1/vector.h"
6 #include "diagonal.h"
7
8 typedef std::function<IterationsResult(const SolverSLAE_Iterative*, const
  ↳ MatrixDiagonal& a, const Vector& y, Vector& x)> method_function;
9
10 //-----
11 void testResidual(const MatrixDiagonal& a, const Vector& x, method_function
  ↳ method, real epsilon) {
12     SolverSLAE_Iterative solver;
13     solver.w = 1;
14     solver.start = Vector(a.dimension(), 5);
15     solver.epsilon = epsilon;
16     solver.maxIterations = 1e5;
17
18     Vector y;
19     mul(a, x, y);
20

```

```

21 Vector x1;
22 auto result = method(&solver, a, y, x1);
23
24 Vector y1;
25 mul(a, x1, y1);
26
27 y1.negate();
28 sum(y, y1, y1);
29 real relativeResidual = calcNorm(y1) / calcNorm(y);
30
31 if (relativeResidual == relativeResidual) {
32     CHECK(fabs(relativeResidual - result.relativeResidual) /
33           ↪ relativeResidual < 0.01);
34
35     if (result.iterations < solver.maxIterations) {
36         CHECK(relativeResidual <= epsilon);
37     }
38 }
39
40 //-----
41 //-----
42 //-----
43
44 TEST_CASE("Test residual of methods") {
45     for (int i = 10; i < 100; ++i) {
46         for (int j = 0; j < 3; ++j) {
47             MatrixDiagonal a;
48             auto format = generateRandomFormat(i, intRandom(10,
49           ↪ Diagonal(i).calcDiagonalsCount()));
50             generateDiagonallyDominantMatrix(i, format, intRandom(0, 10) % 2, a);
51
52             Vector x;
53             x.generate(i);
54
55             testResidual(a, x, &SolverSLAE_Iterative::jacobi, 1e-10);
56             testResidual(a, x, &SolverSLAE_Iterative::seidel, 1e-10);
57         }
58     }
59
60 //-----
61 //-----
62 //-----
63
64 int main(int argc, char* const argv[]) {
65     int result = Catch::Session().run(argc, argv);
66
67     system("pause");
68     return result;
69 }

```

## 4 Исследования

### 4.1 Матрица с диагональным преобладанием

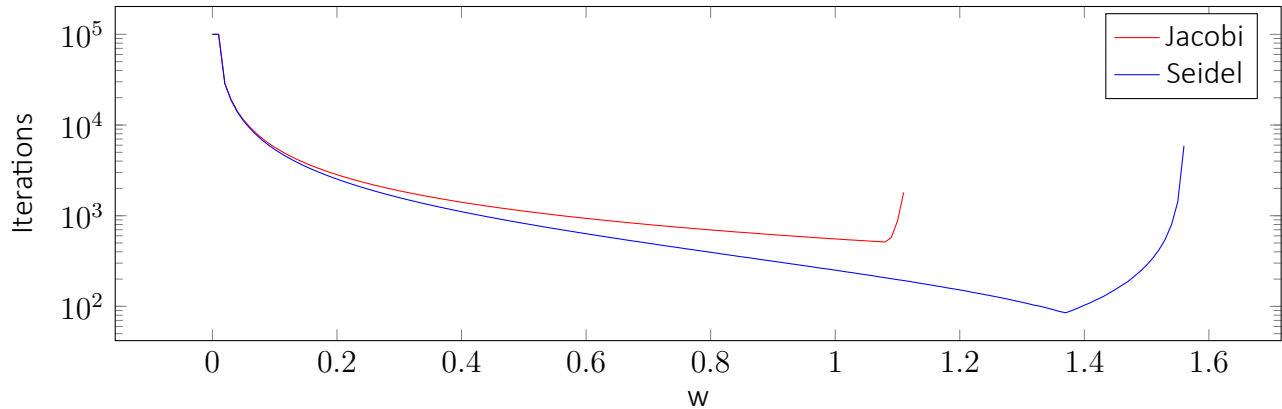
$$A = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -3 & 13 & -4 & 0 & 0 & -4 & 0 & -2 & 0 & 0 \\ 0 & 0 & 7 & -3 & 0 & 0 & -2 & 0 & -2 & 0 \\ 0 & 0 & -3 & 8 & -2 & 0 & 0 & 0 & 0 & -3 \\ -2 & 0 & 0 & -2 & 5 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 \\ -2 & 0 & -4 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 & 0 & 0 & -3 & 7 & -1 & 0 \\ 0 & 0 & -2 & 0 & -4 & 0 & 0 & -3 & 9 & 0 \\ 0 & 0 & 0 & -1 & 0 & -4 & 0 & 0 & -4 & 9 \end{pmatrix}, X = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{pmatrix}, F = \begin{pmatrix} -5 \\ -29 \\ -23 \\ -17 \\ 9 \\ 5 \\ 28 \\ 14 \\ 31 \\ 26 \end{pmatrix}$$

$$\varepsilon = 10^{-14}, \quad iterations_{max} = 100000, \quad start = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)^T$$

Метод Якоби						Метод Зейделя					
0.00	0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000	1.00e+00 2.00e+00 3.00e+00 4.00e+00 5.00e+00 6.00e+00 7.00e+00 8.00e+00 9.00e+00 1.00e+01	1.00e+00	1.00	100000	0.00	0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000 0.0000000000000000	1.0e+00 2.0e+00 3.0e+00 4.0e+00 5.0e+00 6.0e+00 7.0e+00 8.0e+00 9.0e+00 1.0e+01	1.00e+00	1.00	100000
0.10	0.99999999997312 1.99999999994800 2.99999999994125 3.99999999994142 4.99999999995302 5.99999999994751 6.99999999994902 7.99999999994200 8.99999999994404 9.99999999994262	2.69e-13 5.20e-13 5.88e-13 5.86e-13 4.70e-13 5.25e-13 5.10e-13 5.80e-13 5.60e-13 5.74e-13	9.93e-15	8.54	5678	0.10	0.99999999997365 1.99999999994895 2.99999999994222 3.99999999994245 4.99999999995381 5.99999999994866 6.99999999995008 7.99999999994333 8.99999999994529 9.99999999994351	2.6e-13 5.1e-13 5.8e-13 5.8e-13 4.6e-13 5.1e-13 5.0e-13 5.7e-13 5.5e-13 5.6e-13	9.95e-15	8.36	5381
0.20	0.99999999997413 1.99999999994982 2.99999999994329 3.99999999994347 4.99999999995453 5.99999999994946 6.99999999995097 7.99999999994413 8.99999999994564 9.99999999994404	2.59e-13 5.02e-13 5.67e-13 5.65e-13 4.55e-13 5.05e-13 4.90e-13 5.59e-13 5.44e-13 5.60e-13	9.96e-15	8.22	2833	0.20	0.99999999997410 1.99999999994977 2.99999999994329 3.99999999994373 4.99999999995479 5.99999999994991 6.99999999995124 7.99999999994458 8.99999999994635 9.99999999994511	2.6e-13 5.0e-13 5.7e-13 5.6e-13 4.5e-13 5.0e-13 4.9e-13 5.5e-13 5.4e-13 5.5e-13	9.93e-15	8.19	2532
0.30	0.99999999997444 1.99999999995042 2.99999999994396 3.99999999994413 4.99999999995506 5.99999999995008 6.99999999995159 7.99999999994476 8.99999999994635 9.99999999994476	2.56e-13 4.96e-13 5.60e-13 5.59e-13 4.49e-13 4.99e-13 4.84e-13 5.52e-13 5.36e-13 5.52e-13	9.86e-15	8.21	1884	0.30	0.99999999997394 1.99999999994948 2.99999999994307 3.99999999994369 4.99999999995488 5.99999999994991 6.99999999995115 7.99999999994467 8.99999999994653 9.99999999994529	2.6e-13 5.1e-13 5.7e-13 5.6e-13 4.5e-13 5.0e-13 4.9e-13 5.5e-13 5.3e-13 5.5e-13	1.00e-14	8.14	1582
0.40	0.99999999997434 1.99999999995017 2.99999999994365 3.99999999994382 4.99999999995488 5.99999999994991 6.99999999995133 7.99999999994449 8.99999999994618 9.99999999994458	2.57e-13 4.98e-13 5.64e-13 5.62e-13 4.51e-13 5.01e-13 4.87e-13 5.55e-13 5.38e-13 5.54e-13	9.90e-15	8.21	1409	0.40	0.99999999997397 1.99999999994955 2.99999999994316 3.99999999994396 4.99999999995506 5.99999999995026 6.99999999995142 7.99999999994511 8.99999999994689 9.99999999994582	2.6e-13 5.0e-13 5.7e-13 5.6e-13 4.5e-13 5.0e-13 4.9e-13 5.5e-13 5.3e-13 5.4e-13	9.99e-15	8.10	1107
0.50	0.99999999997422 1.99999999994995 2.99999999994347 3.99999999994369 4.99999999995470 5.99999999994973 6.99999999995115 7.99999999994422 8.99999999994582 9.99999999994440	2.58e-13 5.00e-13 5.65e-13 5.63e-13 4.53e-13 5.03e-13 4.88e-13 5.58e-13 5.42e-13 5.56e-13	9.94e-15	8.21	1124	0.50	0.99999999997498 1.99999999995155 2.99999999994547 3.99999999994644 4.99999999995710 5.99999999995257 6.99999999995355 7.99999999994778 8.99999999994955 9.99999999994866	2.5e-13 4.8e-13 5.5e-13 5.4e-13 4.3e-13 4.7e-13 4.6e-13 5.2e-13 5.0e-13 5.1e-13	9.76e-15	7.92	823

0.60	0.99999999997411 1.999999999994975 2.999999999994320 3.999999999994347 4.999999999995453 5.999999999994946 6.999999999995097 7.999999999994404 8.999999999994564 9.999999999994422	2.59e-13 5.02e-13 5.68e-13 5.65e-13 4.55e-13 5.05e-13 4.90e-13 5.60e-13 5.44e-13 5.58e-13	9.89e-15	8.29	934	0.60	0.999999999997568 1.999999999995293 2.999999999994706 3.999999999994831 4.999999999995870 5.999999999995435 6.999999999995524 7.999999999994973 8.999999999995168 9.999999999995097	2.4e-13 4.7e-13 5.3e-13 5.2e-13 4.1e-13 4.6e-13 4.5e-13 5.0e-13 4.8e-13 4.9e-13	9.68e-15	7.69	633
0.70	0.999999999997469 1.999999999995088 2.999999999994449 3.999999999994476 4.999999999995559 5.999999999995062 6.999999999995204 7.999999999994520 8.999999999994689 9.999999999994547	2.53e-13 4.91e-13 5.55e-13 5.52e-13 4.44e-13 4.94e-13 4.80e-13 5.48e-13 5.31e-13 5.45e-13	9.64e-15	8.31	799	0.70	0.999999999997663 1.999999999995477 2.999999999994920 3.999999999995066 4.999999999996056 5.999999999995648 6.999999999995719 7.999999999995230 8.999999999995399 9.999999999995346	2.3e-13 4.5e-13 5.1e-13 4.9e-13 3.9e-13 4.4e-13 4.3e-13 4.8e-13 4.6e-13 4.7e-13	9.53e-15	7.46	497
0.80	0.999999999997469 1.999999999995040 2.999999999994396 3.999999999994413 4.999999999995515 5.999999999995008 6.999999999995159 7.999999999994476 8.999999999994618 9.999999999994493	2.53e-13 4.96e-13 5.60e-13 5.59e-13 4.49e-13 4.99e-13 4.84e-13 5.52e-13 5.38e-13 5.51e-13	9.83e-15	8.23	697	0.80	0.999999999997817 1.999999999995768 2.999999999995257 3.999999999995417 4.999999999996350 5.999999999995977 6.999999999996039 7.999999999995604 8.999999999995772 9.999999999995737	2.2e-13 4.2e-13 4.7e-13 4.6e-13 3.7e-13 4.0e-13 4.0e-13 4.4e-13 4.2e-13 4.3e-13	9.42e-15	6.99	395
0.90	0.999999999997456 1.999999999995062 2.999999999994418 3.999999999994440 4.999999999995532 5.999999999995035 6.999999999995186 7.999999999994502 8.999999999994671 9.999999999994511	2.54e-13 4.94e-13 5.58e-13 5.56e-13 4.47e-13 4.96e-13 4.81e-13 5.50e-13 5.33e-13 5.49e-13	9.85e-15	8.17	618	0.90	0.999999999997934 1.999999999996005 2.999999999995537 3.999999999995723 4.999999999996607 5.999999999996261 6.999999999996296 7.999999999995932 8.999999999996092 9.999999999996074	2.1e-13 4.0e-13 4.5e-13 4.3e-13 3.4e-13 3.7e-13 3.7e-13 4.1e-13 3.9e-13 3.9e-13	9.54e-15	6.43	315
1.00	0.999999999997509 1.999999999995151 2.999999999994529 3.999999999994547 4.999999999995612 5.999999999995133 6.999999999995266 7.999999999994609 8.999999999994760 9.999999999994618	2.49e-13 4.85e-13 5.47e-13 5.45e-13 4.39e-13 4.87e-13 4.73e-13 5.39e-13 5.24e-13 5.38e-13	9.70e-15	8.14	555	1.00	0.999999999998201 1.999999999996521 2.999999999996123 3.999999999996323 4.999999999997087 5.999999999996803 6.999999999996820 7.999999999996536 8.999999999996678 9.999999999996696	1.8e-13 3.5e-13 3.9e-13 3.7e-13 2.9e-13 3.2e-13 3.2e-13 3.5e-13 3.3e-13 3.3e-13	9.02e-15	5.84	251
1.08	0.999999999997538 1.999999999995233 2.999999999994595 3.999999999994640 4.999999999995683 5.999999999995204 6.999999999995355 7.999999999994680 8.999999999994831 9.999999999994706	2.5e-13 4.8e-13 5.4e-13 5.4e-13 4.3e-13 4.8e-13 4.6e-13 5.3e-13 5.2e-13 5.3e-13	9.3e-15	8.35	513	1.08	0.999999999998229 1.999999999996569 2.999999999996190 3.999999999996430 4.999999999997176 5.999999999996909 6.999999999996900 7.999999999996643 8.999999999996803 9.999999999996856	1.8e-13 3.4e-13 3.8e-13 3.6e-13 2.8e-13 3.1e-13 3.1e-13 3.4e-13 3.2e-13 3.1e-13	9.64e-15	5.30	207
1.10	1.0000000000000113 1.999999999999800 3.0000000000000013 4.0000000000000124 4.999999999999813 6.0000000000000284 6.999999999999893 8.0000000000000053 9.0000000000000053 9.999999999999822	-1.13e-14 2.00e-14 -1.33e-15 -1.24e-14 1.87e-14 -2.84e-14 1.07e-14 -5.33e-15 -5.33e-15 1.78e-14	9.12e-15	0.27	875	1.10	0.9999999999998258 1.9999999999996636 2.9999999999996265 3.9999999999996509 4.9999999999997247 5.9999999999996989 6.9999999999996971 7.9999999999996723 8.9999999999996891 9.9999999999996927	1.7e-13 3.4e-13 3.7e-13 3.5e-13 2.8e-13 3.0e-13 3.0e-13 3.3e-13 3.1e-13 3.1e-13	9.63e-15	5.19	197
						1.20	0.9999999999998664 1.9999999999997420 2.9999999999997149 3.9999999999997380 4.9999999999997948 5.9999999999997771 6.9999999999997735 7.9999999999997611 8.9999999999997744 9.9999999999997797	1.3e-13 2.6e-13 2.9e-13 2.6e-13 2.1e-13 2.2e-13 2.3e-13 2.4e-13 2.3e-13 2.2e-13	8.69e-15	4.28	152
						1.30	0.9999999999998852 1.9999999999997773 2.9999999999997558 3.9999999999997824 4.9999999999998312 5.9999999999998179 6.9999999999998126 7.9999999999998055 8.9999999999998206 9.9999999999998295	1.1e-13 2.2e-13 2.4e-13 2.2e-13 1.7e-13 1.8e-13 1.9e-13 1.9e-13 1.8e-13 1.7e-13	9.30e-15	3.31	111
						1.37	0.9999999999999157 1.9999999999998914 2.9999999999998836 3.9999999999999565 4.9999999999999458 5.9999999999998996 6.9999999999998996 7.9999999999999503 8.9999999999999432 9.9999999999999130	8.4e-14 1.1e-13 1.2e-13 4.4e-14 5.4e-14 1.0e-13 1.0e-13 5.0e-14 5.7e-14 8.7e-14	9.10e-15	1.49	85

						1.40	1.000000000000024 1.999999999999885 3.000000000000351 3.999999999999760 4.999999999999698 5.999999999999920 7.000000000000480 8.000000000000018 8.999999999999876 10.000000000000000	-2.4e-15 1.2e-14 -3.5e-14 2.4e-14 3.0e-14 8.0e-15 -4.8e-14 -1.8e-15 1.2e-14 0.0e+00	7.35e-15	0.51	103
						1.50	0.999999999999558 1.999999999999600 3.000000000000373 4.000000000000036 4.999999999999520 5.999999999999396 7.000000000000524 8.000000000000355 8.999999999999840 9.999999999999556	4.4e-14 4.0e-14 -3.7e-14 -3.6e-15 4.8e-14 6.0e-14 -5.2e-14 -3.6e-14 1.6e-14 4.4e-14	8.49e-15	0.79	285



## 4.2 Матрица с обратным знаком внедиагональных элементов

$$B = \begin{pmatrix} 9 & 4 & 0 & 1 & 0 & 0 & 0 & 3 & 0 & 0 \\ 4 & 8 & 0 & 0 & 2 & 0 & 0 & 0 & 2 & 0 \\ 0 & 3 & 9 & 1 & 0 & 3 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 9 & 3 & 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 & 5 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 4 & 0 & 3 & 8 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 2 & 0 & 3 & 10 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 2 & 7 & 1 \\ 0 & 0 & 3 & 0 & 0 & 0 & 3 & 0 & 1 & 7 \end{pmatrix}, X = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{pmatrix}, F = \begin{pmatrix} 45 \\ 48 \\ 75 \\ 44 \\ 97 \\ 59 \\ 100 \\ 148 \\ 113 \\ 109 \end{pmatrix}$$

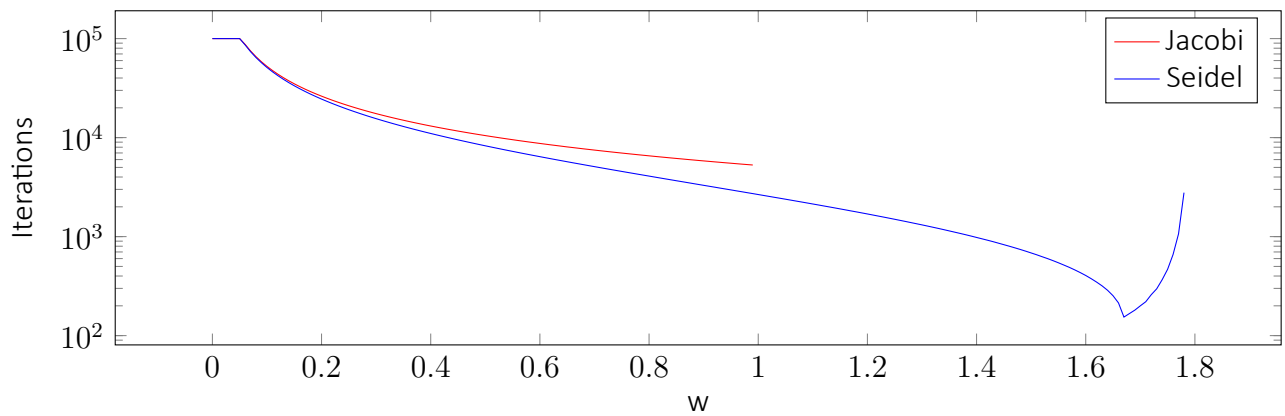
$$\varepsilon = 10^{-14}, \quad iterations_{max} = 100000, \quad start = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)^T$$

Метод Якоби						Метод Зейделя					
0.00	0.000000000000000	1.00e+00	1.00e+00	1.00	100000	0.00	0.000000000000000	1.0e+00	1.00e+00	1.00	100000
	0.000000000000000	2.00e+00					0.000000000000000	2.0e+00			
	0.000000000000000	3.00e+00					0.000000000000000	3.0e+00			
	0.000000000000000	4.00e+00					0.000000000000000	4.0e+00			
	0.000000000000000	5.00e+00					0.000000000000000	5.0e+00			
	0.000000000000000	6.00e+00					0.000000000000000	6.0e+00			
	0.000000000000000	7.00e+00					0.000000000000000	7.0e+00			
	0.000000000000000	8.00e+00					0.000000000000000	8.0e+00			
	0.000000000000000	9.00e+00					0.000000000000000	9.0e+00			
	0.000000000000000	1.00e+01					0.000000000000000	1.0e+01			
0.10	0.999999999748445	2.52e-11	9.99e-15	461.37	52448	0.10	0.999999999750441	2.5e-11	9.99e-15	457.30	51082
	2.000000000272502	-2.73e-11					2.000000000270322	-2.7e-11			
	2.999999999713296	2.87e-11					2.999999999715627	2.8e-11			
	4.000000000295977	-2.96e-11					4.000000000293552	-2.9e-11			
	4.999999999709344	2.91e-11					4.999999999711697	2.9e-11			
	6.000000000290692	-2.91e-11					6.000000000288320	-2.9e-11			
	6.999999999705302	2.95e-11					6.999999999707736	2.9e-11			
	8.000000000289546	-2.90e-11					8.000000000287201	-2.9e-11			
	8.999999999708287	2.92e-11					8.999999999710614	2.9e-11			
	10.000000000291998	-2.92e-11					10.000000000289617	-2.9e-11			

0.20	0.999999999751141 2.000000000269584 2.999999999716391 4.000000000292735 4.999999999712452 6.000000000287601 6.999999999708482 8.000000000286509 8.999999999711324 10.000000000288924	2.49e-11 -2.70e-11 2.84e-11 -2.93e-11 2.88e-11 -2.88e-11 2.92e-11 -2.87e-11 2.89e-11 -2.89e-11	9.98e-15	456.98	26225	0.20	0.999999999751906 2.000000000268678 2.999999999717359 4.000000000291758 4.999999999713474 6.000000000286517 6.999999999709601 8.000000000285354 8.999999999712532 10.000000000287717	2.5e-11 -2.7e-11 2.8e-11 -2.9e-11 2.9e-11 -2.9e-11 2.9e-11 -2.9e-11 2.9e-11 -2.9e-11	1.00e-14	454.29	24500
0.30	0.999999999749128 2.000000000271765 2.999999999714091 4.000000000295106 4.999999999710134 6.000000000289937 6.999999999706111 8.000000000288818 8.999999999708997 10.000000000291251	2.51e-11 -2.72e-11 2.86e-11 -2.95e-11 2.90e-11 -2.90e-11 2.94e-11 -2.89e-11 2.91e-11 -2.91e-11	9.99e-15	459.80	17472	0.30	0.999999999755919 2.000000000264286 2.999999999722000 4.000000000286953 4.999999999718163 6.000000000281757 6.999999999714468 8.000000000280593 8.999999999717382 10.000000000282867	2.4e-11 -2.6e-11 2.8e-11 -2.9e-11 2.8e-11 -2.8e-11 2.9e-11 -2.8e-11 2.8e-11 -2.8e-11	9.97e-15	447.84	15542
0.40	0.999999999749905 2.000000000270926 2.999999999714961 4.000000000294218 4.999999999711005 6.000000000289040 6.999999999706999 8.000000000287912 8.999999999709903 10.000000000290363	2.50e-11 -2.71e-11 2.85e-11 -2.94e-11 2.89e-11 -2.89e-11 2.93e-11 -2.88e-11 2.90e-11 -2.90e-11	1.00e-14	458.23	13103	0.40	0.999999999755989 2.000000000260898 2.999999999725584 4.000000000283276 4.999999999721778 6.000000000278071 6.999999999718225 8.000000000276934 8.999999999721130 10.000000000279083	2.4e-11 -2.6e-11 2.7e-11 -2.8e-11 2.8e-11 -2.8e-11 2.8e-11 -2.8e-11 2.8e-11 -2.8e-11	9.98e-15	441.87	11012
0.50	0.999999999750855 2.000000000269904 2.999999999716045 4.000000000293081 4.999999999712124 6.000000000287947 6.999999999708136 8.000000000286811 8.999999999710987 10.000000000289262	2.49e-11 -2.70e-11 2.84e-11 -2.93e-11 2.88e-11 -2.88e-11 2.92e-11 -2.87e-11 2.89e-11 -2.89e-11	9.88e-15	461.69	10482	0.50	0.999999999762602 2.000000000256919 2.999999999729794 4.000000000278932 4.999999999726050 6.000000000273754 6.999999999722657 8.000000000272617 8.999999999725553 10.000000000274643	2.4e-11 -2.6e-11 2.7e-11 -2.8e-11 2.7e-11 -2.7e-11 2.8e-11 -2.7e-11 2.7e-11 -2.7e-11	1.00e-14	434.20	8267
0.60	0.999999999751421 2.000000000269287 2.999999999716702 4.000000000292371 4.999999999712772 6.000000000287272 6.999999999708828 8.000000000286171 8.999999999711662 10.000000000288587	2.49e-11 -2.69e-11 2.83e-11 -2.92e-11 2.87e-11 -2.87e-11 2.91e-11 -2.86e-11 2.88e-11 -2.89e-11	9.97e-15	456.64	8734	0.60	0.999999999767676 2.000000000251359 2.999999999735665 4.000000000272884 4.999999999732019 6.000000000267715 6.999999999728812 8.000000000266560 8.999999999731681 10.000000000268496	2.3e-11 -2.5e-11 2.6e-11 -2.7e-11 2.7e-11 -2.7e-11 2.7e-11 -2.7e-11 2.7e-11 -2.7e-11	9.97e-15	425.61	6421
0.70	0.999999999750842 2.000000000269909 2.999999999716036 4.000000000293081 4.999999999712097 6.000000000287956 6.999999999708136 8.000000000286846 8.999999999711005 10.000000000289297	2.49e-11 -2.70e-11 2.84e-11 -2.93e-11 2.88e-11 -2.88e-11 2.92e-11 -2.87e-11 2.89e-11 -2.89e-11	9.94e-15	459.33	7484	0.70	0.999999999775033 2.000000000243299 2.999999999744165 4.000000000264126 4.999999999740625 6.000000000259011 6.999999999737659 8.000000000257909 8.999999999740510 10.000000000259668	2.2e-11 -2.4e-11 2.6e-11 -2.6e-11 2.6e-11 -2.6e-11 2.6e-11 -2.6e-11 2.6e-11 -2.6e-11	9.95e-15	412.64	5092
0.80	0.999999999750737 2.000000000270037 2.999999999715921 4.000000000293205 4.999999999711990 6.000000000288081 6.999999999707994 8.000000000286988 8.999999999710880 10.000000000289386	2.49e-11 -2.70e-11 2.84e-11 -2.93e-11 2.88e-11 -2.88e-11 2.92e-11 -2.87e-11 2.89e-11 -2.89e-11	9.99e-15	457.18	6547	0.80	0.999999999784299 2.000000000233187 2.999999999754836 4.000000000253113 4.999999999751452 6.000000000248104 6.999999999748779 8.000000000247002 8.999999999751576 10.000000000248601	2.2e-11 -2.3e-11 2.5e-11 -2.5e-11 2.5e-11 -2.5e-11 2.5e-11 -2.5e-11 2.5e-11 -2.5e-11	9.91e-15	396.99	4087
0.90	0.999999999751336 2.000000000269380 2.999999999716600 4.000000000292504 4.999999999712665 6.000000000287361 6.999999999708704 8.000000000286260 8.999999999711573 10.000000000288676	2.49e-11 -2.69e-11 2.83e-11 -2.93e-11 2.87e-11 -2.87e-11 2.91e-11 -2.86e-11 2.88e-11 -2.89e-11	9.95e-15	457.88	5819	0.90	0.999999999792958 2.000000000223714 2.999999999764833 4.000000000242828 4.999999999761586 6.000000000237881 6.999999999759179 8.000000000236771 8.999999999761933 10.000000000238192	2.1e-11 -2.2e-11 2.4e-11 -2.4e-11 2.4e-11 -2.4e-11 2.4e-11 -2.4e-11 2.4e-11 -2.4e-11	9.96e-15	378.68	3297
0.99	0.999999999751349 2.000000000269420 2.999999999716618 4.000000000292522 4.999999999712701 6.000000000287415 6.999999999708731 8.000000000286313 8.999999999711573 10.000000000288729	2.5e-11 -2.7e-11 2.8e-11 -2.9e-11 2.9e-11 -2.9e-11 2.9e-11 -2.9e-11 2.9e-11 -2.9e-11	9.9e-15	461.89	5289	0.99	0.999999999806718 2.000000000208731 2.999999999780660 4.000000000226503 4.999999999776000 6.000000000221787 6.999999999775557 8.000000000220712 8.999999999778186 10.000000000221902	1.9e-11 -2.1e-11 2.2e-11 -2.3e-11 2.2e-11 -2.2e-11 2.2e-11 -2.2e-11 2.2e-11 -2.2e-11	9.88e-15	356.25	2720
						1.00	0.999999999806523 2.000000000208935 2.999999999780425 4.000000000226716 4.999999999777405 6.000000000221974 6.999999999775380 8.000000000220890 8.999999999778026 10.000000000222062	1.9e-11 -2.1e-11 2.2e-11 -2.3e-11 2.2e-11 -2.2e-11 2.2e-11 -2.2e-11 2.2e-11 -2.2e-11	9.98e-15	352.86	2661

						1.10	0.999999999823919 2.000000000189981 2.999999999800377 4.000000000206155 4.999999999797637 6.000000000201643 6.999999999796030 8.000000000200586 8.999999999798543 10.000000000201545	1.8e-11 -1.9e-11 2.0e-11 -2.1e-11 2.0e-11 -2.0e-11 2.0e-11 -2.0e-11 2.0e-11 -2.0e-11	9.89e-15	323.59	2136
						1.20	0.999999999845336 2.000000000166689 2.999999999824927 4.000000000180869 4.999999999822533 6.000000000176659 6.999999999821370 8.000000000175664 8.999999999823750 10.000000000176321	1.5e-11 -1.7e-11 1.8e-11 -1.8e-11 1.8e-11 -1.8e-11 1.8e-11 -1.8e-11 1.8e-11 -1.8e-11	9.77e-15	286.99	1694
						1.30	0.999999999868533 2.000000000141469 2.999999999851510 4.000000000153442 4.999999999849436 6.000000000149640 6.999999999848859 8.000000000148717 8.999999999850981 10.000000000149054	1.3e-11 -1.4e-11 1.5e-11 -1.5e-11 1.5e-11 -1.5e-11 1.5e-11 -1.5e-11 1.5e-11 -1.5e-11	9.79e-15	242.59	1314
						1.40	0.999999999892694 2.000000000115188 2.999999999879177 4.000000000124878 4.999999999877529 6.000000000121450 6.999999999877476 8.000000000120615 8.999999999879368 10.000000000120615	1.1e-11 -1.2e-11 1.2e-11 -1.2e-11 1.2e-11 -1.2e-11 1.2e-11 -1.2e-11 1.2e-11 -1.2e-11	9.87e-15	195.50	981
						1.50	0.999999999922587 2.000000000082725 2.999999999913363 4.000000000089670 4.999999999912159 6.000000000086713 6.999999999912701 8.000000000085993 8.999999999914326 10.000000000085638	7.7e-12 -8.3e-12 8.7e-12 -9.0e-12 8.8e-12 -8.7e-12 8.7e-12 -8.6e-12 8.6e-12 -8.6e-12	9.88e-15	139.64	684
						1.60	0.999999999954424 2.000000000048197 2.999999999949667 4.000000000052127 4.999999999948992 6.000000000049862 6.999999999950093 8.000000000049276 8.999999999951275 10.000000000048566	4.6e-12 -4.8e-12 5.0e-12 -5.2e-12 5.1e-12 -5.0e-12 5.0e-12 -4.9e-12 4.9e-12 -4.9e-12	9.61e-15	82.83	405
						1.67	1.00000000007827 1.99999999992648 3.00000000007390 3.99999999992268 5.00000000007505 5.99999999993525 7.00000000006084 7.99999999993765 9.00000000005471 9.99999999994689	-7.8e-13 7.4e-13 -7.4e-13 7.7e-13 -7.5e-13 6.5e-13 -6.1e-13 6.2e-13 -5.5e-13 5.3e-13	7.87e-15	13.93	154
						1.70	1.000000000032723 1.999999999963636 3.000000000038507 3.999999999960201 5.000000000039044 5.999999999960316 7.000000000040501 7.999999999960476 9.000000000040181 9.999999999959677	-3.3e-12 3.6e-12 -3.9e-12 4.0e-12 -3.9e-12 4.0e-12 -4.1e-12 4.0e-12 -4.0e-12 4.0e-12	4.71e-15	132.54	200





## 5 Выводы

Исследования показали, что для различных матриц необходим различный параметр релаксации, и что иногда он может лежать за допустимыми пределами (как это было для метода Якоби, где  $w = 1.08$ ). График зависимости числа итераций от параметра релаксации имеет непростой вид, что сильно затрудняет его выбор без перебора всех вариантов.

Так же было оценено число обусловленности:  $cond(A) > 1.49$ ,  $cond(B) > 13.93$ .