

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ М.В.ЛОМОНОСОВА  
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

---

Компьютерный практикум по учебному курсу  
"Введение в численные методы"  
Задание №1

Отчет  
о выполненном задании  
студента 203 учебной группы факультета ВМК МГУ  
Журавского Максима Игоревича

Москва, 2017

# Оглавление

<b>1</b>	<b>Введение</b>	<b>2</b>
<b>2</b>	<b>Алгоритмы решения</b>	<b>3</b>
2.1	Метод Гаусса . . . . .	3
2.2	Метод верхней релаксации . . . . .	3
<b>3</b>	<b>Программа</b>	<b>5</b>
<b>4</b>	<b>Тестирование</b>	<b>22</b>
<b>5</b>	<b>Заключение</b>	<b>26</b>

# Глава 1

## Введение

Целью данной работы является реализация численных методов нахождения:

- решения заданных систем линейных алгебраических уравнений методами Гаусса, в т.ч. методом Гаусса с выбором главного элемента, и методом верхней релаксации.
- определителя заданных матриц
- матрицы, обратной данной

Кроме того, в работе исследуются вопросы устойчивости метода Гаусса при больших размерах матрицы коэффициентов и скорости сходимости к точному решению задачи итераций метода верхней релаксации при изменении итерационного параметра  $\omega$ .

## Глава 2

# Алгоритмы решения

### 2.1 Метод Гаусса

Решения системы линейных алгебраических уравнений методом Гаусса производится в два этапа, именуемые "прямым" и "обратным" ходом, в результате которых из соответствующей исследуемой системе линейных алгебраических уравнений расширенной матрицы находится решение системы. В результате "прямого хода" матрица коэффициентов, входящая в расширенную матрицу, путем линейных преобразований строк и их перестановок последовательно приводится к верхнему треугольному виду. На этапе "обратного хода" выполняется восстановление решения путем прохода по строкам матрицы в обратном направлении.

В отличие от классического метода Гаусса в метода Гаусса с выбором главного элемента на каждой итерации выбирается максимальный из всех элементов матрицы, что позволяет уменьшить погрешность вычислений. Более подробно оба метода описаны в [1].

Задача вычисления определителя сводится к задаче приведения исследуемой матрицы к верхнему треугольному виду и последующим вычислением произведения её диагональных элементов. Для приведения матрицы к диагональному виду используется классический метод Гаусса.

Задача нахождения обратной матрицы решается с помощью метода Гаусса-Жордана. Над дополненной матрицей, составленной из столбцов исходной матрицы и единичной того же порядка, производятся преобразования метода Гаусса, в результате которых исходная матрица принимает вид единичной матрицы, а на месте единичной образуется матрица, обратная исходной. Теоретическое обоснование может быть найдено в [2].

### 2.2 Метод верхней релаксации

Метод верхней релаксации является стационарным итерационным методом, в котором каждый следующий вектор приближения вычисляется по формуле:

$$(D + \omega T_n) \frac{y_{k+1} - y_k}{\omega} + Ay_k = f$$

где

$\omega$  - итерационный параметр,

$y_k$  -  $k$ -й вектор приближения,

$y_{k+1}$  -  $(k + 1)$ -й вектор приближения,

$A$  - матрица коэффициентов,

$f$  - правая часть системы,

$T_{\text{н}}$  - нижняя диагональная матрица матрицы  $A$ ,

$D$  - матрица диагональных элементов матрицы  $A$ .

По теореме Самарского [3] для положительно определенных матриц  $A$  метод верхней релаксации сходится, если  $0 < \omega < 2$ . Однако, оптимальное значение  $\omega$  в каждом случае находится экспериментально и зависит от матрицы  $A$ . В данной реализации метода допускается возможность изменения итерационного параметра с помощью флагов компиляции. Значением по умолчанию является  $\omega = \frac{4}{3}$ .

## Глава 3

# Программа

В качестве языка программирования системы был выбран язык С ввиду его гибкости и высоко производительности. Программа реализована модульно. Основная часть программы реализована в файле **main.c**. Исходный код модулей, отвечающих за реализацию метода Гаусса, метода Гаусса с выбором главного члена и метода верхней релаксации, расположен в файлах **gauss.c**, **modified-gauss.c** и **iteration.c** соответственно. Функции из приложения №2 находятся в файле **functions.c**. Ниже приведено содержание каждого из файлов.

### main.c:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <string.h>
5 #include <math.h>
6
7 /* Constants */
8 typedef enum {
9     SOR, GAUSS, MODIFIED
10 } Method;
11
12 #ifndef OMEGA
13     #define OMEGA 1.3333333333333333
14 #endif
15 #define PRECISION 0.00000001
16
17
18 /* Prototypes */
19 extern double **matrix_create(unsigned n, unsigned m);
20 extern double **matrix_read(unsigned n);
21 extern double *matrix_read_vector(unsigned n);
22 extern int matrix_print(double **matrix, unsigned n);
23 extern int matrix_print_vector(double *vector, unsigned n);
24 extern void matrix_destroy(double **matrix, unsigned n);
25 extern double matrix_determinant(double **matrix, unsigned n);
26 extern double **matrix_inverse(double **matrix, unsigned n);
27
28 extern double *matrix_gauss_solve(double **matrix, const double *f,
29     unsigned n);
29 extern double *matrix_modified_solve(double **matrix, const double *f,
30     unsigned n);
```

```

30 extern double *matrix_iteration_solve(double **a, const double *f, unsigned
    n, double omega, const double *start, double precision);
31
32 extern unsigned functions_init(unsigned task);
33 extern double **matrix_fill();
34 extern double (*matrix_fill_vector)(double x);
35
36
37 /*
38  * Entry Point.
39  */
40 int main(int argc, char *argv[]) {
41     // read matrix
42     unsigned n;
43     double **matrix, *f;
44
45     // parse options
46     if (argc == 2) {
47         // use standard input stream
48         scanf("%u", &n);
49         if ((matrix = matrix_read(n)) == NULL || (f = matrix_read_vector(n)
50 ) == NULL) {
51             fprintf(stderr, "> error: reading failed with message: %s\n",
52 strerror(errno));
53             exit(1);
54         }
55     } else if (argc == 4) {
56         // use formula
57         unsigned long formula = strtoul(argv[2], NULL, 10);
58         if (errno != 0 || formula > 4) {
59             fprintf(stderr, "> error: invalid option\n");
60             exit(2);
61         }
62         double x = atof(argv[3]);
63         n = functions_init((unsigned) formula);
64         matrix = matrix_fill();
65         f = matrix_fill_vector(x);
66     } else {
67         fprintf(stderr, "> error: wrong number arguments: %s gauss|mod|sor
68 [formula x]\n",
69         argv[0]);
70         exit(2);
71     }
72
73     // choose method
74     Method method;
75     double start[n];
76     if (strcmp(argv[1], "gauss") == 0) {
77         // Gaussian elimination method
78         method = GAUSS;
79     } else if (strcmp(argv[1], "mod") == 0) {
80         // Modified Gaussian elimination method
81         method = MODIFIED;
82     } else if (strcmp(argv[1], "sor") == 0) {
83         // SOR method
84         method = SOR;
85     }
86     // create first approximation vector
87     for (unsigned i = 0; i < n; i++) {

```

```

84         start[i] = 0;
85     }
86 } else {
87     fprintf(stderr, "> error: unknown method\n");
88     exit(2);
89 }
90
91 // print processed matrix
92 printf("> matrix:\n");
93 matrix_print(matrix, n);
94 putchar('\n');
95
96 // calculate determinant
97 double determinant = matrix_determinant(matrix, n);
98 printf("> determinant:\n %.10g\n", determinant);
99 putchar('\n');
100
101 // process if possible if possible
102 if (determinant == 0) {
103     printf("> inverse matrix:\n does not exist\n\n");
104     printf("> solution:\n cannot be found using Gaussian elimination
method\n\n");
105 } else {
106     // calculate inverse matrix
107     double **inverse;
108     if ((inverse = matrix_inverse(matrix, n)) == NULL) {
109         fprintf(stderr, "> error: inversion failed\n");
110         exit(1);
111     }
112     printf("> inverse matrix:\n");
113     matrix_print(inverse, n);
114     matrix_destroy(inverse, n);
115     putchar('\n');
116
117 // calculate solution using Gaussian elimination method
118 double *solution;
119 switch (method) {
120     case GAUSS:
121         solution = matrix_gauss_solve(matrix, f, n);
122         break;
123     case MODIFIED:
124         solution = matrix_modified_solve(matrix, f, n);
125         break;
126     case SOR:
127         solution = matrix_iteration_solve(matrix, f, n, OMEGA,
start, PRECISION);
128         break;
129     default:
130         fprintf(stderr, "> error: unknown method option\
nterminationg...\n");
131         exit(1);
132 }
133
134 if (solution == NULL) {
135     fprintf(stderr, "> error: failed to calculate solution: %s\n",
strerror(errno));
136     exit(1);
137 }

```



```

138
139     printf("> solution:\n");
140     matrix_print_vector(solution, n);
141     putchar('\n');
142
143     // free memory
144     free(solution);
145 }
146
147 // free memory
148 free(f);
149 matrix_destroy(matrix, n);
150
151 // completion message
152 printf("> program sucessfully finished!\n");
153 }

```

### gauss.c:

```

1 #include <stdlib.h>
2 #include <errno.h>
3 #include <stdio.h>
4 #include <math.h>
5 #include <stdbool.h>
6
7 int matrix_forward(double **matrix, unsigned n, unsigned m, bool to_swap);
8 void matrix_back(double **matrix, unsigned n, unsigned m);
9 void matrix_normalize(double **matrix, unsigned n, unsigned m);
10
11 /*
12  * Function allocates memory for a matrix of size n * m on the heap and
13  * returns a handle to it.
14  */
15 double **matrix_create(unsigned n, unsigned m) {
16     double **matrix;
17     if ((matrix = malloc(n * sizeof(matrix[0]))) == NULL) {
18         // failed to allocate memory
19         return NULL;
20     }
21     for (unsigned i = 0; i < n; i++) {
22         if ((matrix[i] = malloc(m * sizeof(matrix[i][0]))) == NULL) {
23             // failed to allocate memory
24             for (unsigned j = 0; j < i; j++) {
25                 free(matrix[j]);
26             }
27             free(matrix);
28             return NULL;
29         }
30     }
31     return matrix;
32 }
33
34 /*
35  * Function releases memory occupied by a matrix of size n * n.
36  */
37 void matrix_destroy(double **matrix, unsigned n) {
38     for (unsigned i = 0; i < n; i++) {

```

```

38     free(matrix[i]);
39 }
40 free(matrix);
41 }
42
43 /*
44  * Function reads a matrix of size n * n and returns a handle to it.
45  */
46 double **matrix_read(unsigned n) {
47     double **matrix;
48     if ((matrix = matrix_create(n, n)) == NULL) {
49         return NULL;
50     }
51     for (unsigned i = 0; i < n; i++) {
52         for (unsigned j = 0; j < n; j++) {
53             if (scanf("%lf", &matrix[i][j]) == EOF) {
54                 // failed to read
55                 matrix_destroy(matrix, n);
56                 errno = EINVAL;
57                 return NULL;
58             }
59         }
60     }
61     return matrix;
62 }
63
64 /*
65  * Function reads a matrix of size n * 1 and returns a handle to it.
66  */
67 double *matrix_read_vector(unsigned n) {
68     double *vector;
69     if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
70         // failed to allocate memory
71         return NULL;
72     }
73     for (unsigned i = 0; i < n; i++) {
74         if (scanf("%lf", &vector[i]) == EOF) {
75             // failed to read
76             free(vector);
77             errno = EINVAL;
78             return NULL;
79         }
80     }
81     return vector;
82 }
83
84 /*
85  * Function prints a matrix of size n * n.
86  */
87 int matrix_print(double **matrix, unsigned n) {
88     for (unsigned i = 0; i < n; i++) {
89         for (unsigned j = 0; j < n; j++) {
90             // substitute negative zero with zero
91             if (matrix[i][j] == 0) {
92                 matrix[i][j] = +0.0;
93             }
94             // print matrix row
95             if (printf("%15.10g ", matrix[i][j]) == 0) {

```

```

96         // failed to print
97         return -1;
98     }
99 }
100 // put 'end of line' at the end
101 if (putchar('\n') == EOF) {
102     // failed to print
103     return -1;
104 }
105 }
106 return 0;
107 }
108
109 /*
110  * Function prints a matrix of size n * 1.
111  */
112 int matrix_print_vector(double *vector, unsigned n) {
113     for (unsigned i = 0; i < n; i++) {
114         // substitute negative zero with zero
115         if (vector[i] == 0) {
116             vector[i] = +0.0;
117         }
118         // print vector
119         if (printf("%15.10g ", vector[i]) == 0) {
120             // failed to print
121             return -1;
122         }
123     }
124     // put 'end of line' at the end
125     if (putchar('\n') == EOF) {
126         return -1;
127     }
128     return 0;
129 }
130
131 /*
132  * Function solves matrix equation  $Ax = f$  using Gaussian Elimination method
133  */
134 double *matrix_gauss_solve(double **matrix, const double *f, unsigned n) {
135     // create augmented matrix
136     double **aug = matrix_create(n, n + 1);
137     if (aug == NULL) {
138         return NULL;
139     }
140     for (unsigned i = 0; i < n; i++) {
141         for (unsigned j = 0; j < n; j++) {
142             aug[i][j] = matrix[i][j];
143         }
144         aug[i][n] = f[i];
145     }
146
147     // perform forward elimination, normalization and back substitution
148     matrix_forward(aug, n, n + 1, true);
149     matrix_normalize(aug, n, n + 1);
150     matrix_back(aug, n, n + 1);
151
152     // calculate the result

```

```

153     double *result;
154     if ((result = malloc(n * sizeof(result[0]))) == NULL) {
155         matrix_destroy(aug, n);
156         return NULL;
157     }
158     for (unsigned i = 0; i < n; i++) {
159         result[i] = aug[i][n];
160     }
161
162     // free memory
163     matrix_destroy(aug, n);
164     return result;
165 }
166
167 /*
168  * Function returns the number of the row with the greatest primary element
169  */
170 unsigned matrix_find_greatest(double **matrix, unsigned current, unsigned n) {
171     unsigned result = current;
172     for (unsigned j = current + 1; j < n; j++) {
173         if (matrix[j][current] != 0 && (!matrix[result][current] || fabs(
174             matrix[result][current]) < fabs(matrix[j][current]))) {
175             result = j;
176         }
177     }
178     return result;
179 }
180
181 /*
182  * Function swap specified rows in a matrix.
183  */
184 void matrix_swap_rows(double **matrix, unsigned i, unsigned j) {
185     double *temp = matrix[i];
186     matrix[i] = matrix[j];
187     matrix[j] = temp;
188 }
189
190 /*
191  * Function subtracts current line from the following ones.
192  * retruns -1 if division by 0 was about to occur.
193  */
194 int matrix_subtract(double **matrix, unsigned current, unsigned n, unsigned
195 m) {
196     for (unsigned i = current + 1; i < n; i++) {
197         if (matrix[current][current] == 0) {
198             return -1;
199         }
200         double multiplier = matrix[i][current] / matrix[current][current];
201         if (multiplier != 0) {
202             for (unsigned j = current; j < m; j++) {
203                 matrix[i][j] -= multiplier * matrix[current][j];
204             }
205         }
206     }
207     return 0;
208 }

```

```

207
208 /*
209  * Function performs Forward Elimination of the augmented matrix  $n * m$  ( $n$ 
     $\geq m$ ).
210  * Flag signal whether rows should be swapped or not.
211 */
212 int matrix_forward(double **matrix, unsigned n, unsigned m, bool to_swap) {
213     for (unsigned i = 0; i < n; i++) {
214         // subtract greatest from the rest
215         if (to_swap) {
216             unsigned greatest = matrix_find_greatest(matrix, i, n);
217             if (i != greatest) {
218                 matrix_swap_rows(matrix, i, greatest);
219             }
220         }
221         if (matrix_subtract(matrix, i, n, m) == -1) {
222             return -1;
223         }
224     }
225
226     return 0;
227 }
228
229 /*
230  * Function normalizes upper triangular  $n * m$  matrix.
231 */
232 void matrix_normalize(double **matrix, unsigned n, unsigned m) {
233     for (unsigned i = 0; i < n; i++) {
234         for (unsigned j = i + 1; j < m; j++) {
235             matrix[i][j] /= matrix[i][i];
236         }
237         matrix[i][i] = 1;
238     }
239 }
240
241 /*
242  * Function performs back substitution.
243 */
244 void matrix_back(double **matrix, unsigned n, unsigned m) {
245     for (unsigned i = n - 1; i > 0; i--) {
246         for (unsigned prev = 0; prev < i; prev++) {
247             // subtract current row from previous
248             double multiplier = matrix[prev][i];
249             if (multiplier == 0) {
250                 continue;
251             }
252             for (unsigned j = i; j < m; j++) {
253                 matrix[prev][j] -= matrix[i][j] * multiplier;
254             }
255         }
256     }
257 }
258
259 /*
260  * Function returns determinant of an  $n * n$  matrix.
261 */
262 double matrix_determinant(double **matrix, unsigned n) {
263

```

```

264 // calculate upper triangular matrix
265 double **aug = matrix_create(n, n);
266 if (aug == NULL) {
267     // failed to allocate matrix
268     return 0;
269 }
270 for (unsigned i = 0; i < n; i++) {
271     for (unsigned j = 0; j < n; j++) {
272         aug[i][j] = matrix[i][j];
273     }
274 }
275 if (matrix_forward(aug, n, n, false) == -1) {
276     matrix_destroy(aug, n);
277     return 0;
278 }
279
280 // calculate determinant
281 double result = 1;
282 for (unsigned i = 0; i < n; i++) {
283     result *= aug[i][i];
284 }
285
286 // free memory
287 matrix_destroy(aug, n);
288 return result;
289 }
290
291 /*
292 * Function returns a handle to the inverse of a given n * n matrix.
293 */
294 double **matrix_inverse(double **matrix, unsigned n) {
295     // create augmented matrix
296     double **aug = matrix_create(n, 2 * n);
297     if (aug == NULL) {
298         return NULL;
299     }
300     for (unsigned i = 0; i < n; i++) {
301         for (unsigned j = 0; j < n; j++) {
302             aug[i][j] = matrix[i][j];
303         }
304         for (unsigned j = n; j < n * 2; j++) {
305             aug[i][j] = (i == (j - n) ? 1 : 0);
306         }
307     }
308
309     // perform forward elimination and back substitution
310     matrix_forward(aug, n, 2 * n, false);
311     matrix_normalize(aug, n, 2 * n);
312     matrix_back(aug, n, 2 * n);
313
314     // copy the result
315     double **result = matrix_create(n, n);
316     if (result == NULL) {
317         return NULL;
318     }
319     for (unsigned i = 0; i < n; i++) {
320         for (unsigned j = 0; j < n; j++) {
321             result[i][j] = aug[i][j + n];

```

```

322     }
323 }
324
325 // free memory
326 matrix_destroy(aug, n);
327
328 return result;
329 }

```

### modified-gauss.c:

```

1 #include <stdlib.h>
2 #include <math.h>
3 #include <string.h>
4 #include <stdio.h>
5 #include <stdbool.h>
6
7 extern double **matrix_create(unsigned n, unsigned m);
8 extern void matrix_destroy(double **matrix, unsigned n);
9 extern void matrix_swap_rows(double **matrix, unsigned i, unsigned j);
10
11 void matrix_modified_forward(double **matrix, unsigned *transform, unsigned
    n, unsigned m);
12 void matrix_modified_back(double **matrix, const unsigned *transform,
    unsigned n, unsigned m);
13 void matrix_modified_normalize(double **matrix, const unsigned *transform,
    unsigned n, unsigned m);
14
15 typedef struct {
16     unsigned i;
17     unsigned j;
18 } Point;
19
20 /*
21  * Function solves matrix equation  $Ax = f$  using modified Gaussian
    Elimination method.
22  */
23 double *matrix_modified_solve(double **matrix, const double *f, unsigned n)
24 {
25     // create augmented matrix and a transformation vector
26     double **aug = matrix_create(n, n + 1);
27     if (aug == NULL) {
28         return NULL;
29     }
30     for (unsigned i = 0; i < n; i++) {
31         for (unsigned j = 0; j < n; j++) {
32             aug[i][j] = matrix[i][j];
33         }
34         aug[i][n] = f[i];
35     }
36     unsigned transform[n];
37     for (unsigned i = 0; i < n; i++) {
38         transform[i] = i;
39     }
40
41     // perform forward elimination and normalization
42     matrix_modified_forward(aug, transform, n, n + 1);

```

```

42
43 // calculate result
44 double *result;
45 if ((result = malloc(n * sizeof(result[0]))) == NULL) {
46     matrix_destroy(aug, n);
47     return NULL;
48 }
49 for (unsigned i = 0; i < n; i++) {
50     // find effective x
51     unsigned col = 0;
52     for (unsigned j = 0; j < n; j++) {
53         if (transform[j] == i) {
54             col = j;
55             break;
56         }
57     }
58     // write out the result
59     result[col] = aug[i][n];
60 }
61
62 // free memory
63 matrix_destroy(aug, n);
64 return result;
65 }
66
67 /* Function returns the number of the row and column of the greatest
   element */
68 Point matrix_modified_find_greatest(double **matrix, const unsigned *
   transform,
69     unsigned current, unsigned n) {
70     Point result = {current, 0};
71     bool isFound = false;
72     for (unsigned i = current; i < n; i++) {
73         for (unsigned j = 0; j < n; j++) {
74             if (transform[j] < current) {
75                 continue;
76             }
77             if (matrix[i][j] != 0 &&
78                 (!isFound || fabs(matrix[result.i][result.j]) < fabs(matrix[i][j]
79 ))) ) {
80                 result.i = i;
81                 result.j = j;
82                 isFound = true;
83             }
84         }
85     }
86     return result;
87 }
88
89 /* Function performs Forward Elimination of the augmented matrix n * m */
90 void matrix_modified_forward(double **matrix, unsigned *transform, unsigned
   n, unsigned m) {
91     for (unsigned i = 0; i < n; i++) {
92         // find greatest from others and put it to front
93         Point greatest = matrix_modified_find_greatest(matrix, transform, i
94 , n);
95         if (i != greatest.i) {
96             matrix_swap_rows(matrix, i, greatest.i);

```



```

95     }
96     if (i != transform[greatest.j]) {
97         unsigned col = 0;
98         for (unsigned j = 0; j < n; j++) {
99             if (transform[j] == i) {
100                 col = j;
101                 break;
102             }
103         }
104         unsigned temp = transform[greatest.j];
105         transform[greatest.j] = transform[col];
106         transform[col] = temp;
107     }
108
109     // divide current row
110     for (unsigned j = 0; j < m; j++) {
111         if (j == greatest.j) {
112             continue;
113         }
114         matrix[i][j] /= matrix[i][greatest.j];
115     }
116     matrix[i][greatest.j] = 1;
117
118     // reduce other rows
119     const unsigned col = greatest.j;
120     for (unsigned row = 0; row < n; row++) {
121         if (row == i) {
122             continue;
123         }
124         double multiplier = matrix[row][col];
125         if (multiplier != 0) {
126             for (unsigned j = 0; j < m; j++) {
127                 matrix[row][j] -= multiplier * matrix[i][j];
128             }
129         }
130     }
131 }
132 }

```

### iteration.c:

```

1 #include <stdlib.h>
2 #include <math.h>
3 #include <string.h>
4 #include <stdio.h>
5
6 extern double **matrix_create(unsigned n, unsigned m);
7 extern void matrix_destroy(double **matrix, unsigned n);
8
9 /*
10  * Function normalizes lower triangular n * m matrix.
11  */
12 void normalize_reversed(double **matrix, unsigned n, unsigned m) {
13     for (unsigned i = 0; i < n; i++) {
14         for (unsigned j = 0; j < i; j++) {
15             matrix[i][j] /= matrix[i][i];
16         }
17     }
18 }

```

```

17     matrix[i][m - 1] /= matrix[i][i];
18     matrix[i][i] = 1;
19 }
20 }
21
22 /*
23  * Function performs back substitution.
24 */
25 void back_reversed(double **matrix, unsigned n, unsigned m) {
26     for (unsigned i = 0; i < n; i++) {
27         for (unsigned next = i + 1; next < n; next++) {
28             // subtract current row from previous
29             double multiplier = matrix[next][i];
30             if (multiplier == 0) {
31                 continue;
32             }
33             for (unsigned j = 0; j < m; j++) {
34                 matrix[next][j] -= matrix[i][j] * multiplier;
35             }
36         }
37     }
38 }
39
40
41 /*
42  * Function calculates next element of the iteration sequence.
43 */
44 void matrix_iteration_next(const double **a, double **b, const double *f,
45     unsigned n, double omega, double *current) {
46     // create new right part in the augmented B matrix
47     for (unsigned i = 0; i < n; i++) {
48         b[i][n] = f[i];
49         for (unsigned j = i; j < n; j++) {
50             if (i == j) {
51                 b[i][n] -= (1 - 1 / omega) * a[i][j] * current[i];
52             } else {
53                 b[i][n] -= a[i][j] * current[j];
54             }
55         }
56     }
57     // solve matrix equation with the new f vector: Bx = f
58     normalize_reversed(b, n, n + 1);
59     back_reversed(b, n, n + 1);
60
61     // copy the result
62     for (unsigned i = 0; i < n; i++) {
63         current[i] = b[i][n];
64     }
65 }
66
67 /*
68  * Function calculates vector difference.
69 */
70 double matrix_vector_diff(const double *a, const double *b, unsigned n) {
71     double result = 0;
72     for (unsigned i = 0; i < n; i++) {
73         result += fabs(a[i] - b[i]);

```

```

74     }
75
76     return result;
77 }
78
79 /*
80  * Functions calculates an estimate solution of the matrix equation
81  *  $Ax = f$  using successive over-relaxation method. Function returns a handle
82  * to the result.
83  */
84 double *matrix_iteration_solve(const double **a, const double *f, unsigned
85                                n, double omega, const double *start, double precision) {
86     // allocate vectors for current and previous elements of iteration
87     // sequence
88     double *current, *previous;
89     if ((current = malloc(sizeof(current[0]) * n)) == NULL) {
90         return NULL;
91     }
92     if ((previous = malloc(sizeof(previous[0]) * n)) == NULL) {
93         free(current);
94         return NULL;
95     }
96
97     // create matrix B
98     double **aug;
99     if ((aug = matrix_create(n, n + 1)) == NULL) {
100         free(current);
101         free(previous);
102         return NULL;
103     }
104
105     // start iteration process
106     unsigned iterationCount = 0;
107     memcpy(current, start, n * sizeof(start[0]));
108     do {
109         // construct B matrix
110         for (unsigned i = 0; i < n; i++) {
111             for (unsigned j = 0; j < n; j++) {
112                 if (i < j) {
113                     aug[i][j] = 0;
114                 } else if (i == j) {
115                     aug[i][j] = (1 / omega) * a[i][j];
116                 } else {
117                     aug[i][j] = a[i][j];
118                 }
119             }
120         }
121
122         // move result of the previous iteration and perform iteration
123         memcpy(previous, current, n * sizeof(current[0]));
124         matrix_iteration_next(a, aug, f, n, omega, current);
125         iterationCount++;
126     } while (matrix_vector_diff(current, previous, n) > precision);
127
128     printf("> log: %d iterations made\n", iterationCount);
129
130     // release memory
131     matrix_destroy(aug, n);

```

```

129     free(previous);
130
131     return current;
132 }

```

### functions.c:

```

1 #include <stdlib.h>
2 #include <math.h>
3 #include <assert.h>
4
5 /*
6  * Functions for tasks from Appendix 2.
7  * Tasks 1-4.
8  */
9 extern double **matrix_create(unsigned n, unsigned m);
10
11 static unsigned M = 0;
12 static unsigned n = 0;
13 double *(*matrix_fill_vector)(double x) = NULL;
14
15 static double *f1(double x) {
16     double *vector;
17     if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
18         return NULL;
19     }
20     for (unsigned i = 0; i < n; i++) {
21         vector[i] = n * exp(x / i) * cos(x);
22     }
23
24     return vector;
25 }
26
27 static double *f2(double x) {
28     double *vector;
29     if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
30         return NULL;
31     }
32     for (unsigned i = 0; i < n; i++) {
33         vector[i] = fabs(x - (double) n / 10) * i * sin(x);
34     }
35
36     return vector;
37 }
38
39 static double *f3(double x) {
40     double *vector;
41     if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
42         return NULL;
43     }
44     for (unsigned i = 0; i < n; i++) {
45         vector[i] = x * exp(x / i) * cos(x / i);
46     }
47
48     return vector;
49 }
50

```

```

51 static double *f4(double x) {
52     double *vector;
53     if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
54         return NULL;
55     }
56     for (unsigned i = 0; i < n; i++) {
57         vector[i] = n * exp(x / i) * cos(x);
58     }
59
60     return vector;
61 }
62
63 /*
64  * Function initializes the module for using formula-determined matrices.
65  */
66 unsigned functions_init(unsigned task) {
67     switch(task) {
68         case 1:
69             M = 1;
70             n = 50;
71             matrix_fill_vector = f1;
72             break;
73         case 2:
74             M = 2;
75             n = 40;
76             matrix_fill_vector = f2;
77             break;
78         case 3:
79             M = 3;
80             n = 30;
81             matrix_fill_vector = f3;
82             break;
83         case 4:
84             M = 4;
85             n = 100;
86             matrix_fill_vector = f4;
87             break;
88         default:
89             assert(0);
90     }
91
92     return n;
93 }
94
95 /*
96  * Function fills the matrix A using the specified formula.
97  */
98 double **matrix_fill() {
99     const double q = 1.001 - 2 * M * 0.001;
100
101     double **matrix;
102     if ((matrix = matrix_create(n, n)) == NULL) {
103         return NULL;
104     }
105     for (unsigned i = 0; i < n; i++) {
106         for (unsigned j = 0; j < n; j++) {
107             if (i == j) {
108                 matrix[i][j] = pow(q - 1, i + j);

```

```

109         } else {
110             matrix[i][j] = pow(q, i + j) + 0.1 * (j - i);
111         }
112     }
113 }
114
115 return matrix;
116 }

```

## Глава 4

### Тестирование

Тестирование правильности решения систем линейных алгебраических уравнений программой проводилось как с помощью проверки результатов работы программы путем подстановки их в исходную систему, так и с помощью сравнения результатов работ двух методов. Тестирование нахождения обратных матриц и определителя проверялось вручную для матриц, размером меньше  $3 \times 3$ , и с помощью пакета Maple для матриц больше порядка. Ниже приведены условия тестов в виде расширенных матриц систем алгебраических уравнений и результаты работы программы. Первые три теста взяты из варианта №13 тестовых заданий.

Тест #1

$$\left( \begin{array}{cccc|c} 3 & -2 & 2 & -2 & 8 \\ 2 & -1 & 2 & 0 & 4 \\ 2 & 1 & 4 & 8 & -1 \\ 1 & 3 & -6 & 2 & 3 \end{array} \right)$$

Результат работы метода Гаусса:

> determinant:

24

> inverse matrix:

-0.5 1.333333333 -0.1666666667 0.1666666667

-5 8.666666667 -1.333333333 0.3333333333

-2 3.5 -0.5 0

1.75 -3.166666667 0.5833333333 -0.08333333333

> solution:

2 -3 -1.5 0.5

Результат работы метода Гаусса с выбором главного элемента:

> solution:

2 -3 -1.5 0.5

Результат работы метода верхней релаксации:

> log: 235 iterations made

> solution:

-nan -nan -nan -nan

Тест #2

$$\left(\begin{array}{cccc|c} 2 & 3 & 1 & 2 & 4 \\ 4 & 3 & 1 & 1 & 5 \\ 1 & -7 & -1 & -2 & 7 \\ 2 & 5 & 1 & 1 & 1 \end{array}\right)$$

Результат работы метода Гаусса:

> determinant:

2

> inverse matrix:

-1 2 -1 -2

-1 1.5 -1 -1.5

8 -14.5 9 16.5

-1 3 -2 -4

> solution:

-3 -5 39 -7

Результат работы метода Гаусса с выбором главного элемента:

> solution:

-3 -5 39 -7

Результат работы метода верхней релаксации:

> log: 271 iterations made

> solution:

-nan -nan -nan -nan

Тест #3

$$\left(\begin{array}{cccc|c} 1 & -1 & 1 & -1 & 0 \\ 4 & -1 & 0 & -1 & 0 \\ 2 & 1 & -2 & 1 & 0 \\ 5 & 1 & 0 & -4 & 0 \end{array}\right)$$

Результат работы метода Гаусса:

> determinant: 0 > inverse matrix: does not exist > solution: cannot be found using Gaussian elimination method

Тест #4

$$\left(\begin{array}{ccc|c} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 3 \\ 1 & 0 & 0 & 4 \end{array}\right)$$

Результат работы метода Гаусса:

> determinant:

-1

> inverse matrix:

0 0 1



```
0 1 0
1 0 -1
> solution:
4 3 -2
```

Результат работы метода Гаусса с выбором главного элемента:

```
> solution:
4 3 -2
```

Результат работы метода верхней релаксации:

```
> log: 1 iterations made
> solution:
2.6666666667 4 -nan
```

Тест #5

$$\left(\begin{array}{cc|c} 1 & 1 & 0 \\ 1 & 2 & 1 \end{array}\right)$$

Результат работы метода Гаусса:

```
> determinant:
1
> inverse matrix:
2 -1
-1 1
> solution:
-1 1
```

Результат работы метода Гаусса с выбором главного элемента:

```
> solution:
-1 1
```

Результат работы метода верхней релаксации:

```
> log: 19 iterations made
> solution:
-1.0000000003 1.0000000001
```

Тест #6

$$\left(\begin{array}{ccc|c} 3 & -1 & 1 & 3 \\ 1 & 1 & 1 & 5 \\ 4 & -1 & 4 & 5 \end{array}\right)$$

Результат работы метода Гаусса:

```
> determinant:
10
> inverse matrix:
0.5 0.3 -0.2
0 0.8 -0.2
-0.5 -0.1 0.4
```

```
> solution:
2 3 0
```

Результат работы метода Гаусса с выбором главного элемента:

```
> solution:
2 3 0
```

Результат работы метода верхней релаксации:

```
> log: 25 iterations made
> solution:
2.0000000001 3.0000000001 0
```

Тест #7

$$\left( \begin{array}{ccc|c} 2 & -1 & 1 & 6 \\ 1 & 2 & -1 & -1 \\ 1 & -1 & 2 & 11 \end{array} \right)$$

Результат работы метода Гаусса:

```
> determinant:
6
> inverse matrix:
0.5 0.1666666667 -0.1666666667
-0.5 0.5 0.5
-0.5 0.1666666667 0.8333333333
> solution:
1 2 6
```

Результат работы метода Гаусса с выбором главного элемента:

```
> solution:
1 2 6
```

Результат работы метода верхней релаксации:

```
> log: 58 iterations made
> solution:
0.9999999987 2.0000000002 6.0000000002
```

Ввиду того, что лишь в последних двух тестах матрицы были положительно определенными, результаты работы метода верхней релаксации в большинстве других были далеки от истинных. В целом, при  $\omega = \frac{4}{3}$  метод верхней релаксации получал решение, достаточно близкое к точному менее, чем за 100 итераций.

## Глава 5

### Заключение

В ходе работы мной были освоены методы решения систем линейных алгебраических уравнений, а именно: метод верхней релаксации и метод Гаусса. Из результатов легко убедиться в том, что ввиду ограниченной применимости метода верхней релаксации, порой он оказывается неприменимым. Тем не менее, при аккуратном подборе итерационного параметра, он может оказаться даже быстрее метода Гаусса. Преимуществом метода Гаусса же является простота его реализации и широкая применимость в задачах, связанных с исследованием матриц, в том числе нахождением обратной матрицы и детерминанта.

# Литература

- [1] Самарский А.А. Введение в численные методы. СПб., 2009. С. 91.
- [2] Белоусов И.В. Матрицы и определители. Кишинев, 2006. С. 68.
- [3] Самарский А.А. Введение в численные методы. СПб., 2009. С. 102.