Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ
И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра прикладной математики и искусственого интеллекта

Направление подготовки: 01.03.04 – Прикладная математика

ОТЧЁТ

По дисциплине «Численные методы»
на тему:
«Система линейных алгебраических уравнений»

Выполнил:
студент группы 09-222
Фаррахова. Л.Ф.
Проверил:
ассистент Глазырина О.В.

Казань, 2024 год

Содержание

1 Постановка задачи

Решить систему линейных алгебраических уравнений:

$$\begin{cases} (a_1 + a_2 + h^2 g_1) y_1 - a_2 y_2 = f_1 h^2, \\ \quad \dots \quad \dots \quad \dots \quad \dots \\ -a_i y_{i-1} + (a_i + a_{i+1} + h^2 g_i) y_i - a_{i+1} y_{i+1} = f_i h^2, \\ \quad \dots \quad \dots \quad \dots \quad \dots \\ (a_{n-1} + a_n + h^2 g_{n-1}) y_{n-1} - a_{n-1} y_{n-2} = f_{n-1} h^2. \end{cases} \tag{1}$$

Здесь $a_i = p(ih),\ g_i = q(ih),\ f_i = f(ih),\ f(x) = -(p(x)u'(x))' + q(x)u(x)$, $h = 1/n,\ p,\ q,\ u$ – заданные функции.

Для этого использовать метод прогонки и итерационные методы:

1. метод Зейделя.
2. метод нижней релаксации.
3. метод наискорейшего спуска.

Продолжать вычисления в итерационных методах, пока не выполнится условие:

$$\max_{1 \le i \le n-1} \left| r_i^k \right| \le \varepsilon,$$

r – вектор невязки, ε – заданное число.

Исходные данные: $n = 10,\ n = 50,\ \varepsilon = h^3,\ u(x) = x^\alpha (1-x)^\beta$, $p(x) = 1 + x^\gamma,\ g(x) = x + 1,\ \alpha = 4,\ \beta = 1,\ \gamma = 1.$

Сравнить результаты вычислений, составив соответсвтенные таблицы и найти наилучший из методов решения.

2 Ход работы

2.1 Метод прогонки

Метод прогонки состоит из двух этапов: прямой ход (определение прогоночных коэффициентов), обратных ход (вычисление неизвестных y_i).

Основным преимуществом является экономичность, ведь метод максимально использует структуру исходной системы.

К недостаткам же можно отнести то, что с каждой итерацией накапливается ошибка округления.

Прямой ход метода заключается в нахождении прогоночных коэффициентов:

$$\begin{cases} \alpha_1 = \dfrac{a_1}{a_0 + a_1 + h^2 g_1}, \\ \alpha_{i+1} = \dfrac{a_{i+1}}{a_i + a_{i+1} + h^2 g_i - \alpha_i a_i}, \quad i = \overline{2, n-1}; \end{cases} \tag{2}$$

$$\begin{cases} \beta_1 = \dfrac{f_0 h^2}{a_0 + a_1 + h^2 g_1}, \\ \beta_{i+1} = \dfrac{f_i h^2 + \beta_i a_i}{a_i + a_{i+1} + h^2 g_i - \alpha_i a_i}, \quad i = \overline{2, n-1}; \end{cases} \tag{3}$$

Обратный ход метода - вычисление формул для нахождения неизвестных:

$$\begin{cases} y_n = \beta_{n+1}; \\ y_i = \alpha_{i+1} y_{i+1} + \beta_{i+1}, \quad i = \overline{n-1, 0}; \end{cases} \tag{4}$$

Формулы (2-4) являются методом Гаусса, записанным применительно трёхдиагональной системы уравнений. Метод может быть реализован только в случае, когда в формулах (3) и (4) все знаменатели отличны от нуля, то есть условие выполняется, когда матрица системы (2) имеет диагональное препобладание.

Проделаем вычисления и составим таблицу для $n = 10$ (Таблица 1), для $n = 50$ (Таблица 2):

ih	y_i	$u(ih)$	$\lvert y_i - u(ih)\rvert$
0,0	0,000000	0,000000	0,000000
0,1	0,014011	0,000090	0,013921
0,2	0,027929	0,001280	0,026649
0,3	0,044160	0,005670	0,038490
0,4	0,065086	0,015360	0,049726
0,5	0,091907	0,031250	0,060657
0,6	0,123480	0,051840	0,071640
0,7	0,155141	0,072030	0,083111
0,8	0,177525	0,081920	0,095605
0,9	0,175383	0,065610	0,109773
1,0	0,126400	0,000000	0,126400

Таблица 1 - значения метода прогонки для $n = 10$

ih	y_i	$u(ih)$	$\lvert y_i - u(ih)\rvert$
0.1	0.002538	0.000090	0.002448
0.2	0.005974	0.001280	0.004694
0.3	0.012450	0.005670	0.006780
0.4	0.024101	0.015360	0.008741
0.5	0.041867	0.031250	0.010617
0.6	0.064297	0.051840	0.012457
0.7	0.086357	0.072030	0.014327
0.8	0.098232	0.081920	0.016312
0.9	0.084129	0.065610	0.018519
1.0	0.021080	0.000000	0.021080

Таблица 2 - значения метода прогонки для $n = 50$

2.2 Метод Якоби

Для больших систем вида $Ax = b$ предпочтительнее оказываются итерационные методы. Основная идея данных методов состоит в построении последовательности векторов x^k, $k = 1, 2, \ldots$, сходящейся к решению исходной системы. За приближенное решение принимается вектор x^k при достаточно большом k.

Будем считать, что все диагональные элементы матрицы из полной системы $Ax = b$ отличны от нуля. Представим эту систему, разрешая каждое уравнение относительно переменной, стоящей на главной диагонали:

$$x_i = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j - \sum_{j=i+1}^{n} \frac{a_{ij}}{a_{ii}} x_j + \frac{b_i}{a_{ii}}, \quad i = \overline{1, n}. \tag{5}$$

Выберем некоторое начальное приближение $x^0 = (x_1^0, x_2^0, \ldots, x_n^0)^T$. Построим последовательность векторов $x^1, x^2, \ldots$, определяя вектор x^{k+1} по уже найденному вектору x^k при помощи соотношения:

$$x_i^{k+1} = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^k - \sum_{j=i+1}^{n} \frac{a_{ij}}{a_{ii}} x_j^k + \frac{b_i}{a_{ii}}, \quad i = \overline{1, n}. \tag{6}$$

Формула (6) определяют итерационный метод решения системы (5), называемый методом Якоби или методом простой итерации.

Запишем этот метод для нашей системы:

$$y_i^{k+1} = -\sum_{j=1}^{i-1} \frac{a_j}{a_i + a_{i+1} + h^2 g_i} y_j^k \; - \; \sum_{j=i+1}^{n} \frac{a_j}{a_i + a_{i+1} + h^2 g_i} y_j^k \; + \; \frac{f_i h^2}{a_i + a_{i+1} + h^2 g_i},$$

$$i = \overline{1, n-1}; \tag{7}$$

Вычисления продолжаем, пока не выполнится условие:

$$\max |r_i^k| \leq \varepsilon,$$

где r^k – вектор невязки для k-той итерации $r^k = Ay^k - f, \quad \varepsilon = h^3$.

Составим таблицы вычисленных результатов для $n = 10$, для $n = 50$, в которых будем сравнивать значения метода прогонки и метода Якоби для точки i, $i = \overline{0, n-1}$, найдём модуль их разности и значение k, при котором была достигнута необходимая точность.

i	y_i	y_i^k	$\lvert y_i - y_i^k \rvert$	k
0	0,014011	0,011380	0,002631	36
1	0,027929	0,023304	0,004625	36
2	0,044160	0,038220	0,005940	36
3	0,065086	0,058498	0,006587	36
4	0,091907	0,085288	0,006619	36
5	0,123480	0,117355	0,006125	36
6	0,155141	0,149921	0,005219	36
7	0,177525	0,173497	0,004027	36
8	0,175383	0,172703	0,002680	36
9	0,126400	0,125099	0,001301	36

Таблица 3 - значения метода Якоби для $n = 10$

i	y_i	y_i^k	$\lvert y_i - y_i^k \rvert$	k
0	0,000509	0,000408	0,000100	955
4	0,002538	0,002067	0,000471	955
8	0,005120	0,004337	0,000783	955
12	0,009344	0,008320	0,001024	955
16	0,016397	0,015209	0,001188	955
20	0,027169	0,025898	0,001271	955
24	0,041867	0,040593	0,001274	955
28	0,059618	0,058415	0,001203	955
32	0,078085	0,077019	0,001066	955
36	0,093067	0,092191	0,000876	955
40	0,098113	0,097467	0,000646	955
45	0,076309	0,075983	0,000327	955
49	0,021080	0,021016	0,000064	955

Таблица 4 - значения метода Якоби для $n = 50$

2.3 Метод нижней релаксации

Во многих ситуациях существенного ускорения сходимости можно добиться за счет введения так называемого итерационного параметра. Рассмотрим итерационный процесс:

$$x_i^{k+1} = (1-\omega)x_i^k + \omega\Bigg(-\sum_{j=1}^{i-1}\frac{a_{ij}}{a_{ii}}x_j^{k+1} - \sum_{j=i+1}^{n}\frac{a_{ij}}{a_{ii}}x_j^k + \frac{b_i}{a_{ii}}\Bigg),$$

$$i = 1, 2, \ldots, n, \quad k = 0, 1, \ldots \ . \tag{8}$$

Этот метод называется методом релаксации – одним из наиболее эффективных и широко используемых итерационных методов для решения систем линейных алгебраических уравнений. Значение ω – называется релаксационным параметром. При $\omega = 1$ метод переходит в метод Зейделя. При $\omega \in (0, 1)$ – метод нижней релаксации, при $\omega \in (1, 2)$ – это метод верхней релаксации.

Преобразуем формулу (9) относительно нашей системы:

$$y_i^{k+1} = (1-\omega)y_i^k + \omega\Bigg(-\sum_{j=1}^{i-1}\frac{a_j}{a_i + a_{i+1} + h^2 g_i}y_j^k \ - \sum_{j=i+1}^{n}\frac{a_j}{a_i + a_{i+1} + h^2 g_i}y_j^k \ + \ \frac{f_i h^2}{a_i + a_{i+1} + h^2 g_i}\Bigg),$$

$$i = \overline{1, n-1}; \tag{9}$$

Параметр ω следует выбирать так, чтобы метод релаксации сходился наиболее быстро. Нужно отметить, что оптимальный параметр для метода верхней релаксации лежит вблизи 0,8. Заполним таблицы, в которых приведём значения параметра ω и количство итераций k:

ω	k
0.1	63
0.2	49
0.3	56
0.4	40
0.5	32
0.6	26
0.7	24
0.8	27
0.9	34

Таблица 5 - значения ω и соответствующие значения k для $n = 10$

ω	k
0.02	892
0.14	721
0.30	452
0.38	228
0.54	190
0.62	141
0.78	125
0.84	117
0.90	144
0.94	159

Таблица 6 - значения ω и соответствующие значения k для $n = 50$

Для вычислений выберем $\omega = 0,84$. Составим таблицы результатов для $n = 10$, $n = 50$, в которых будем сравнивать значения метода прогонки и метода верхней релаксации для точки i, $i = \overline{0, n-1}$, найдём модуль их разности и значение k:

i	y_i	y_i^k	$\lvert y_i - y_i^k \rvert$	k
0	0,014011	0,013805	0,000207	27
1	0,027929	0,027484	0,000444	27
2	0,044160	0,043500	0,000660	27
3	0,065086	0,064323	0,000762	27
4	0,091907	0,091294	0,000613	27
5	0,123480	0,123466	0,000014	27
6	0,155141	0,155153	0,000012	27
7	0,177525	0,177554	0,000029	27
8	0,175383	0,175418	0,000034	27
9	0,126400	0,126425	0,000025	27

Таблица 7 - значения метода нижней релаксации для $n = 10$

i	y_i	y_i^k	$\lvert y_i - y_i^k \rvert$	k
0	0,000509	0,000387	0,000122	117
4	0,002538	0,002007	0,000531	117
8	0,005120	0,004300	0,000820	117
12	0,009344	0,008347	0,000997	117
16	0,016397	0,015323	0,001073	117
20	0,027169	0,026103	0,001066	117
24	0,041867	0,040874	0,000993	117
28	0,059618	0,058747	0,000871	117
32	0,078085	0,077368	0,000717	117
36	0,093067	0,092520	0,000547	117
40	0,098113	0,097739	0,000375	117
45	0,076309	0,076137	0,000173	117
49	0,021080	0,021049	0,000032	117

Таблица 8 - значения метода нижней релаксации для $n = 50$

2.4 Метод наискорейшего спуска

Метод наискорейшего спуска для решений систем линейных алгебраических уравнений заключается в итерационном процессе, направленном на минимизацию квадратичной функции ошибки.

x_k - вычислено, тогда

$$x^{k+1} = x^k - \tau r_i^k, i = \overline{1,n}, k = 0, 1, ... \tag{10}$$

$$r^k = Ax^k - b, \tau = \frac{(r^k, r^k)}{(Ar^k, r^k)}, (r^k, t^k) = \sum_{i=1}^{n-1} (r_i^k)^2, (Ar^k)_i = -a_i r_{i-1}^k + (a_i + a_{i+1} + \\ + h^2 g_i) r_i^k - a_{i+1} r_{i+1}^k$$

r_i^k - вектор невязки для конкретного уравненения:

$$r_i^k = -a_i x_{i-1}^k + (a_i + a_{i+1} + h^2 g_i - a_{i+1}^k - f_i h^2) \tag{11}$$

По сравнению с методом Якоби этот метод требует на каждом шаге итераций проведения дополнительной работы по вычислению параметра τ. Вследствие этого происходит адаптация к оптимальной скорости сходимости.

Составим таблицы результатов для $n = 10$, $n = 50$, в которых будем сравнивать значения метода прогонки и метода верхней релаксации для точки i, $i = \overline{0, n-1}$, найдём модуль их разности и значение k:

i	y_i	y_i^k	$\lvert y_i - y_i^k \rvert$	k
0	0,014011	0,012035	0,001977	38
1	0,027929	0,024447	0,003482	38
2	0,044160	0,039702	0,004458	38
3	0,065086	0,060300	0,004785	38
4	0,091907	0,087169	0,004738	38
5	0,123480	0,119300	0,004180	38
6	0,155141	0,151830	0,003310	38
7	0,177525	0,175069	0,002455	38
8	0,175383	0,174025	0,001358	38
9	0,126400	0,125790	0,000609	38

Таблица 9 - значения метода наискорейшего спуска для $n = 10$

i	y_i	y_i^k	$\lvert y_i - y_i^k \rvert$	k
0	0,000509	0,000487	0,000022	2549
4	0,002538	0,002435	0,000103	2549
8	0,005120	0,004951	0,000168	2549
12	0,009344	0,009128	0,000215	2549
16	0,016397	0,016153	0,000243	2549
20	0,027169	0,026916	0,000253	2549
24	0,041867	0,041620	0,000247	2549
32	0,078085	0,077889	0,000196	2549
36	0,093067	0,092909	0,000157	2549
40	0,098113	0,097999	0,000114	2549
44	0,084129	0,084060	0,000069	2549
49	0,021080	0,021067	0,000013	2549

Таблица 10 - значения метода наискорейшего спуска для $n = 50$

3 Выводы

В процессе выполнения работы были изучены методы решения заданной системы линейных алгебраических уравнений метод прогонки, итерационными методами: Якоби, нижней релаксации, наискорейшего спуска.

В результате наилучшим способом показал себя метод верхней релаксации при итерационном параметре $\omega = 1,84$. Данный метод показывается наилучшие результаты вычисления корней системы за наименьшее количество итераций.

4 Листинг программы

```
1
2  using namespace std;
3
4  #include <algorithm>
5  #include <cmath>
6  #include <iostream>
7  #include <vector>
8
9  double f_x(double x) {
10     return -pow(x, 6) + 26 * pow(x, 4) + 4 * pow(x, 3) - 12 * pow(x, 2);
11 }
12
13 double u_x(double x) {
14     return pow(x, 4) * (1 - x);
15 }
16
17 double p_x(double x) {
18    return 1 + x;
19 }
20
21 double q_x(double x) {
22     return 1 + x;
23 }
24
25 vector<double> func(int n) {
26     double h = 1. / n;
27     vector<double> b(n,0);
28     for (int i = 1; i <= n; ++i) {
29         b[i - 1] = f_x(i * h) * pow(h,2);
30     }
31     return b;
32 }
33
34 double calculate_error(vector<double> &x, vector<vector<double>> &A,
    vector<double> &b) {
35
36     vector<double> r = calculate_r(A, b, x);
37     double max_err = 0.0;
38     for (int i = 0; i < r.size(); ++i) {
39         if (abs(r[i]) > max_err)
40             max_err = abs(r[i]);
```

```
41      }
42
43      return max_err;
44  }
45
46  void progonka_method(vector<vector<double>> &A, vector<double> &x, int n,
        vector<double> &b) {
47
48      vector<double> alpha(n + 1), betta(n + 1);
49      double h = 1.0 / n;
50
51      // прямойход
52      alpha[0] = A[0][1] / A[0][0];
53      betta[0] = (b[0]) / A[0][0];
54
55      for (int i = 1; i < n; ++i) {
56          double del = 1.0 / (A[i][i] - alpha[i - 1] * A[i][i - 1]);
57          alpha[i] = A[i][i + 1] * del;
58          betta[i] = (-A[i][i - 1] * betta[i - 1] + b[i]) * del;
59      }
60
61      // обратныйход
62      x[n - 1] = betta[n - 1];
63      for (int i = n - 2; i >= 0; --i) {
64          x[i] = -alpha[i] * x[i + 1] + betta[i];
65      }
66      for (int i = 1; i <= n; ++i) {
67          printf("ih = %4.2lf | y_i = %9.6lf | u(ih) = %8.6lf | |y_i - u(ih
              )| = %8.6lf\n", i * h, x[i - 1], u_x(i * h), abs(x[i - 1] -
              u_x(i * h)));
68      }
69  }
70
71  int Jacobi_method(int n, vector<double> &x, vector<vector<double>> &A,
      vector<double> &b) {
72      double h = 1.0 / n;
73      int k = 0;
74      vector<double> new_x(n, 0.);
75      double error = 1.0;
76
77      while (error > 1.0 / pow(n, 3)) {
78          vector<double> curr_x = x;
79          for (int i = 0; i < n; ++i) {
```

```
            new_x[i] = Jacobi_calculate_new_x(i, new_x, n, A, b);
        }
        x = new_x;
        error = calculate_error(new_x, A, b);
        k++;
    }
    return k;
}

double Jacobi_calculate_new_x(int i, vector<double>& x, int n, vector<
    vector<double>> &A, vector<double> &b) {
    double h = 1.0 / n;
    double sum = 0.0;


    if (i > 0) {
        for (int j = 0; j <= i - 1; ++j) {
            sum += A[i][j] * x[j];
        }
    }
    for (int j = i + 1; j < n + 1; ++j) {
        sum += A[i][j] * x[j];
    }

    return (b[i] - sum) * (1.0 / A[i][i]);
}


int relax_bottom(int n, vector<double>& x, vector<vector<double>>& A,
    vector<double>& b) {
    double h = 1.0 / n;
    double omega = 0.8;
    double error = 1.0;
    int k = 0;
    vector<double> new_x(n, 0.);

    while (error > 1.0 / pow(n, 3)) {
        for (int i = 0; i < n; ++i) {
            new_x[i] = Relax_calculate_new_x(i, new_x, n, A, omega, b);
        }
        x = new_x;
        error = calculate_error(new_x, A, b);
        k++;
```

```
121     }
122     return k;
123 }
124 
125 double Relax_calculate_new_x(int i, vector<double>& x, int n, vector<
    vector<double>> &A, double omega, vector<double> &b) {
126 
127     double sum = 0.0;
128     if (i > 0) {
129         for (int j = 0; j <= i - 1; ++j) {
130             sum += A[i][j] * x[j];
131         }
132     }
133     for (int j = i + 1; j < n + 1; ++j) {
134         sum += A[i][j] * x[j];
135     }
136     double new_x = (b[i] - sum) * (1.0 / A[i][i]);
137     return x[i] + omega * (new_x - x[i]);
138 }
139 
140 
141 int spusk(int n, vector<double>& x, vector<vector<double>>& A,  vector<
    double>& b) {
142     double h = 1.0 / n;
143     int k = 1;
144     vector<double> new_x(n, 0.);
145     double error = 1.0;
146 
147     while (error > 1.0 / pow(n, 3)) {
148         for (int i = 0; i < n; ++i) {
149             new_x[i] = spusk_calculate_new_x(i, new_x, n, A, b);
150         }
151         x = new_x;
152         error = calculate_error(new_x, A, b);
153         k++;
154     }
155 }
156 
157 double spusk_calculate_new_x(int i,  vector<double>& x, int n,  vector<
    vector<double>>& A,  vector<double>& b) {
158     vector<double> r = calculate_r(A, b, x);
159     vector<double> Ar = spusk_calculate_matrix_vector_multiplication(A, r
        );
```

```
160
161     return x[i] -  spusk_calculate_tau(r, Ar) * r[i];
162 }
163
164 vector<double> calculate_r( vector<vector<double>>& A,  vector<double>& b
    ,  vector<double> &x) {
165     vector<double> Ax = spusk_calculate_matrix_vector_multiplication(A, x
        );
166     vector<double> r(Ax.size());
167
168     for (size_t i = 0; i < r.size(); ++i) {
169         r[i] = Ax[i] - b[i];
170     }
171
172     return r;
173 }
174
175 vector<double> spusk_calculate_matrix_vector_multiplication(vector<vector
    <double>>& A,  vector<double> &x) {
176     int n = A.size();
177     int m = x.size();
178     vector<double> result(n, 0.0);
179
180     for (int i = 0; i < n; ++i) {
181         for (int j = 0; j < m; ++j) {
182             result[i] += A[i][j] * x[j];
183         }
184     }
185     return result;
186 }
187
188 double spusk_calculate_tau( vector<double>& r,  vector<double>& Ar) {
189     double a = 0.;
190     double b = 0.;
191     for (size_t i = 0; i < r.size(); ++i) {
192         a += r[i] * r[i];
193         b += Ar[i] * r[i];
194     }
195     if (abs(b) < 1e-10) {
196         return 0.0;
197     }
198     return a * (1.0 / b);
199 }
```

```
200
201
202 vector<vector<double>> create_matrix(int n){
203     double h = 1. / n;
204
205     vector<vector<double>> matrix_res(n + 1, vector<double>(n + 1,0.0));
206
207     for (int i = 0; i < n; ++i) {
208         if (i == 0) {
209             double b = (p_x((i + 1) * h) + p_x((i + 2) * h) + (pow(h,2) *
                    q_x((i + 1) * h)));
210             double c = -p_x((i + 2) * h);
211             matrix_res[i][i] = b;
212             matrix_res[i][i + 1] = c;
213             continue;
214         }
215         if (i == (n - 1)) {
216             double b = (p_x((i + 1) * h) + p_x((i + 2) * h) + (pow(h,2) *
                    q_x((i + 1) * h)));
217             double a = -p_x((i + 1) * h);
218             matrix_res[i][i] = b;
219             matrix_res[i][i - 1] = a;
220             continue;
221         }
222         double a = -p_x((i + 1) * h);
223         double b = (p_x((i + 1) * h) + p_x((i + 2) * h) + (pow(h,2) * q_x
               ((i + 1) * h)));
224         double c = -p_x((i + 2) * h);
225
226         matrix_res[i][i] = b;
227         matrix_res[i][i + 1] = c;
228         matrix_res[i][i - 1] = a;
229     }
230
231     return matrix_res;
232 }
233
234 void m_print(int k, vector<double> &y_i, vector<double> &y_ik) {
235     for (int i = 0; i < y_i.size() - 1; ++i) {
236         printf("ih = %3d | y_i = %9.6lf | y_ik = %9.6lf | |y_i - y_ik| =
               %9.6lf | k = %d\n", i, y_i[i], y_ik[i], abs(y_i[i] - y_ik[i]),
                k);
237     }
```

```
238 }
239
240
241 int main() {
242
243     int n = 10.0;
244
245     std::vector<double> y_i_p(n + 1), y_i_s(n + 1), y_i_r(n + 1), y_i_sp(
        n + 1), b(n);
246     vector<vector<double>> A = create_matrix(n);
247     b = func(n);
248     printf("Прогонка\n");
249     progonka_method(A, y_i_p, n, b);
250     int k_s = Jacobi_method(n, y_i_s, A, b);
251     int k_r = relax_bottom(n, y_i_r, A, b);
252     int k_dec = spusk(n, y_i_sp, A, b);
253     printf("Прогонка - Якоби\n");
254     m_print(k_s, y_i_p, y_i_s);
255     printf("Прогонка - Нижняярелаксация \n");
256     m_print(k_r, y_i_p, y_i_r);
257     printf("Прогонка - Наискорейшийспуск \n");
258     m_print(k_dec, y_i_p, y_i_sp);
259     return 0;
260 }
```