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
Ахметзянов К.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.

Данную систему решить методом прогоки и итерационными методами:

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

Во всех итерационных методах вычисления продолжать до выполнения условия:

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

r – вектор невязки, ε – заданное число.

Исходные данные: $n = 10,\ n = 50,\ \varepsilon = h^3,\ u(x) = x^\alpha (1-x)^\beta$, $p(x) = 1 + x^\gamma,\ g(x) = x + 1,\ \alpha = 1,\ \beta = 1,\ \gamma = 2.$

Для сравнения результатов вычисления составим таблицы и подведём выводы.

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 – максимально использование структуры исходной системы.

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

Реализуем прямой ход метода и найдём прогоночные коэффициенты:

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

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

Обратных ход метода:

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

Формулы (2-4) являются методом Гаусса, записанным применительно трёхдиагональной системы уравнений. Метод может быть реализован только в случае, когда в формулах (2) и (3) все знаменатели отличны от нуля, то есть условие выполняется, когда матрица системы (1) имеет диагональное препобладание.

Проделаем вычисления и составим таблицу для $n = 10$, для $n = 50$:

ih	y_i	$u(ih)$	ϵ
0.10	0.106769	0.090000	0.016769
0.20	0.184376	0.160000	0.024376
0.30	0.235499	0.210000	0.025499
0.40	0.263158	0.240000	0.023158
0.50	0.270334	0.250000	0.020334
0.60	0.259690	0.240000	0.019690
0.70	0.233435	0.210000	0.023435
0.80	0.193281	0.160000	0.033281
0.90	0.140478	0.090000	0.050478
1.00	0.075876	0.000000	0.075876

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

ih	y_i	$u(ih)$	ϵ
0.10	0.101429	0.090000	0.011429
0.20	0.174182	0.160000	0.014182
0.30	0.220545	0.210000	0.010545
0.40	0.243242	0.240000	0.003242
0.50	0.245064	0.250000	0.004936
0.60	0.228573	0.240000	0.011427
0.70	0.195932	0.210000	0.014068
0.80	0.148846	0.160000	0.011154
0.90	0.088567	0.090000	0.001433
1.00	0.015952	0.000000	0.015952

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

2.2 Метод Зейделя

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.

Метод Якоби допускает естественную модификацию: при вычислении x_i^{k+1} будем использовать уже найденные компоненты вектора x^{k+1}, то есть x_1^{k+1}, x_2^{k+1}, $\ldots$, x_{i-1}^{k+1}. В результате приходим к итерационному методу Зейделя:

$$x_i^{k+1} = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^{k+1} - \sum_{j=i+1}^{n} \frac{a_{ij}}{a_{ii}} x_j^k + \frac{b_i}{a_{ii}}, \quad i = \overline{1, n}. \tag{7}$$

Запишем формулу (7) для нашей системы (1):

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

$$i = \overline{1, n-1}; \tag{8}$$

Вычисления продолжаем, пока не выполнится условие:

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

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

Составим таблицы вычесленных результатов для $n = 10$, для $n = 50$, в которых будем сравнивать значения метода прогонки и метода Зейделя 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	ϵ	k
0	0.106769	0.103847	0.002922	37
1	0.184376	0.179133	0.005243	37
2	0.235499	0.228694	0.006805	37
3	0.263158	0.255599	0.007559	37
4	0.270334	0.262779	0.007555	37
5	0.259690	0.252777	0.006913	37
6	0.233435	0.227639	0.005795	37
7	0.193281	0.188898	0.004383	37
8	0.140478	0.137629	0.002849	37
9	0.075876	0.074529	0.001347	37

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

i	y_i	y_i^k	ϵ	k
0	0.022665	0.022564	0.000101	1155
10	0.185482	0.184502	0.000981	1155
20	0.245190	0.243795	0.001395	1155
30	0.223275	0.222033	0.001242	1155
40	0.137813	0.137131	0.000682	1155
49	0.015952	0.015886	0.000065	1155

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

2.3 Метод верхней релаксации

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

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

$$i = 1, 2, \ldots, n, \quad k = 0, 1, \ldots \ . \tag{9}$$

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 $\omega \in (1, 2)$ – это метод верхней релаксации, при $\omega \in (0, 1)$ – метод нижней релаксации. Ясно, что по затратам памяти и объему вычислений на каждом шаге итераций метод релаксации не отличается от метода Зейделя.

Преобразуем формулу (9) относительно нашей системы:

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

$$i = \overline{1, n-1}; \tag{10}$$

Параметр ω следует выбирать так, чтобы метод релаксации сходился наиболее быстро. Нужно отметить, что оптимальный параметр для метода верхней релаксации лежит вблизи 1,8. Заполним таблицы, в которых приведём значения параметра ω и количство итераций k:

ω	k
1.1	31
1.2	25
1.3	20
1.4	17
1.5	13
1.6	11
1.7	16
1.8	22
1.9	53

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

ω	k
1.10	944
1.20	769
1.30	620
1.40	493
1.50	383
1.60	286
1.70	201
1.80	125
1.90	93

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

Для вычислений выберем $\omega = 1.6$ для $n = 10$ и $\omega = 1.90$ для $n = 50$. Составим таблицы результатов. 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	ϵ	k
0	0.106769	0.107336	0.000567	11
1	0.184376	0.185502	0.001126	11
2	0.235499	0.237047	0.001549	11
3	0.263158	0.264915	0.001756	11
4	0.270334	0.272060	0.001726	11
5	0.259690	0.261174	0.001484	11
6	0.233435	0.234526	0.001091	11
7	0.193281	0.193918	0.000637	11
8	0.140478	0.140705	0.000227	11
9	0.075876	0.075852	0.000024	11

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

i	y_i	y_i^k	ϵ	k
0	0.022665	0.022656	0.000008	93
10	0.185482	0.185453	0.000029	93
20	0.245190	0.245161	0.000029	93
30	0.223275	0.223260	0.000016	93
40	0.137813	0.137810	0.000003	93
49	0.015952	0.015952	0.000000	93

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

2.4 Метод наискорейшего спуска

Существуют итерационные методы, позволяющие за счет некоторой дополнительной работы на каждом шаге итераций автоматически настраиваться на оптимальную скорость сходимости. К их числу относятся методы, основанные на замене системы (6) эквивалентной задачей минимизации некоторого функционала.

Опишем итерационный метод наискорейшего спуска. Будем двигаться из точки начального приближения x^0 в направлении наибыстрейшего убывания функционала F, то есть следующее приближение будем разыскивать так: $x^1 = x^0 - \tau \mathrm{grad}F(x^0)$. Формула:

$$F'_{x_i}(x) = 2\sum_{j=1}^{n} a_{ij}x_j - 2b_i; \tag{11}$$

, которая является производной функции $F(x)$ по переменной x_i, показывает, что $\mathrm{grad}F(x^0) = 2(Ax^0 - b)$. Вектор $r_0 = Ax^0 - b$ принято называть невязкой. Для сокращения записей удобно обозначить 2τ вновь через τ. Таким образом, $x^1 = x^0 - \tau r^0$.

Параметр τ выберем так, чтобы значение $F(x^1)$ было минимальным. Получим $F(x^1) = F(x^0 - \tau r^0) = F(x^0) - 2\tau(r^0, r^0) + \tau^2(Ar^0, r^0)$, следовательно, минимум $F(x^1)$ достигается при $\tau = \tau_* = \dfrac{(r^0, r^0)}{(Ar^0, r^0)}$.

Таким образом, мы пришли к следующему итерационнму методу:

$$x^{k+1} = x^k - \tau_* r^k, \quad r^k = Ax^k - b, \quad \tau_* = \frac{(r^k, r^k)}{(Ar^k, r^k)}, \ k = 0, 1, \dots \ . \tag{12}$$

Метод (12) называют методом наискорейшего спуска. По сравнению с методом простой итерации 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	ϵ	k
0	0.106769	0.105365	0.001404	78
1	0.184376	0.181841	0.002535	78
2	0.235499	0.232225	0.003274	78
3	0.263158	0.259567	0.003591	78
4	0.270334	0.266804	0.003530	78
5	0.259690	0.256517	0.003173	78
6	0.233435	0.230798	0.002637	78
7	0.193281	0.191326	0.001955	78
8	0.140478	0.139067	0.001412	78
9	0.075876	0.075415	0.000461	78

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

i	y_i	y_i^k	ϵ	k
0	0.022665	0.022583	0.000081	816
1	0.044129	0.043967	0.000161	816
2	0.064402	0.064163	0.000240	816
3	0.083498	0.083182	0.000316	816
4	0.101429	0.101039	0.000389	816
5	0.118210	0.117750	0.000460	816
6	0.133858	0.133331	0.000527	816
7	0.148390	0.147800	0.000590	816
8	0.161825	0.161176	0.000649	816
9	0.174182	0.173479	0.000703	816
10	0.185482	0.184730	0.000753	816
20	0.245190	0.244231	0.000959	816
30	0.223275	0.222550	0.000725	816
40	0.137813	0.137481	0.000332	816
49	0.015952	0.015923	0.000029	816

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

3 Выводы

В процессе выполнения работы были изучены методы решения заданной системы линейных алгебраических уравнений вида:

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

при помощи:

1. метод прогонки.
2. метод Зейделя.
3. метод верхней релаксации.
4. метод наискорейшего спуска.

После вычисления результатов решения системы линейных алгебраических уравнений можно сделать вывод, что наилучшим методом для решения является метод верхней релаксации при итерационном параметре $\omega = 1.6$ для системы из 10 уравнений и $\omega = 1.90$ для системы из 50 уравнений. Данный метод показывается наилучшие результаты вычисления корней системы за наименьшее количество итераций.

4 Список литературы

1. Глазырина Л.Л., Карчевский М.М. Численные методы: учебное пособие. — Казань: Казан. ун-т, 2012. — 122

2. Глазырина Л.Л.. Практикум по курсу «Численные методы». Решение систем линейных уравнений: учеб. пособие. — Казань: Изд-во Казан. ун-та, 2017. — 52 с.

5 Листинг программы

```cpp
#pragma once

#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

using namespace std;

double f_x(double x) {
  return 3 * pow(x, 2) - 2 * x + 3;
}

double u_x(double x) { return x * (1 - x); }

double p_x(double x) { return 1 + pow(x, 2); }

double q_x(double x) { return 1 + x; }

vector<double> func_vec(int n) {
  double h = 1. / n;
  vector<double> b(n + 1, 0);
  for (int i = 1; i <= n; ++i) {
    b[i - 1] = f_x(i * h) * pow(h, 2);
  }
  return b;
}

vector<double> calculate_mtx_vec_mult(vector<vector<double>> &A,
                                      vector<double> &x) {
  int n = A.size();
  int m = x.size();
  vector<double> result(n, 0.0);

  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      result[i] += A[i][j] * x[j];
    }
  }

  return result;
```

```
}

vector<double> calculate_r(vector<vector<double>> &A, vector<double> &b,
                           vector<double> &x) {
  vector<double> Ax = calculate_mtx_vec_mult(A, x);
  vector<double> r(Ax.size());

  for (size_t i = 0; i < r.size(); ++i) {
    r[i] = Ax[i] - b[i];
  }

  return r;
}

double calculate_error_dec(vector<double> &x, vector<vector<double>> &A,
                           vector<double> &b) {
  vector<double> r = calculate_r(A, b, x);
  double max_err = 0.0;
  for (int i = 0; i < r.size(); ++i) {
    if (abs(r[i]) > max_err) max_err = abs(r[i]);
  }

  return max_err;
}
double calculate_error(vector<double> &new_x, vector<double> &old_x) {
  double max_err = 0.0;
  for (int i = 0; i < old_x.size(); ++i) {
    double err = abs(abs(new_x[i]) - abs(old_x[i]));
    if (err > max_err) max_err = err;
  }

  return max_err;
}

vector<vector<double>> create_matrix(int n) {
  double h = 1. / n;

  vector<vector<double>> matrix_res(n + 1, vector<double>(n + 1, 0.0));

  for (int i = 0; i < n; ++i) {
    if (i == 0) {
      double b = (p_x((i + 1) * h) + p_x((i + 2) * h) +
                  (pow(h, 2) * q_x((i + 1) * h)));
```

```
85        double c = -p_x((i + 2) * h);
86        matrix_res[i][i] = b;
87        matrix_res[i][i + 1] = c;
88        continue;
89      }
90
91      if (i == (n - 1)) {
92        double b = (p_x((i + 1) * h) + p_x((i + 2) * h) +
93                    (pow(h, 2) * q_x((i + 1) * h)));
94        double a = -p_x((i + 1) * h);
95        matrix_res[i][i] = b;
96        matrix_res[i][i - 1] = a;
97        continue;
98      }
99
100     double a = -p_x((i + 1) * h);
101     double b =
102         (p_x((i + 1) * h) + p_x((i + 2) * h) + (pow(h, 2) * q_x((i + 1) *
                h)));
103     double c = -p_x((i + 2) * h);
104
105     matrix_res[i][i] = b;
106     matrix_res[i][i + 1] = c;
107     matrix_res[i][i - 1] = a;
108   }
109
110   return matrix_res;
111 }
112
113 //Алгорит Томаса
114 void progonka_method(vector<vector<double>> &A, vector<double> &x, int n,
115                      vector<double> &b) {
116   vector<double> alpha(n + 1), betta(n + 1);
117
118   double h = 1.0 / n;
119
120   // прямойход
121   alpha[0] = A[0][1] / A[0][0];
122   betta[0] = (b[0]) / A[0][0];
123
124   for (int i = 1; i < n; ++i) {
125     double del = 1.0 / (A[i][i] - alpha[i - 1] * A[i][i - 1]);
126     alpha[i] = A[i][i + 1] * del;
```

```
127     betta[i] = (-A[i][i - 1] * betta[i - 1] + b[i]) * del;
128   }
129
130   // обратныйход
131   x[n - 1] = betta[n - 1];
132   for (int i = n - 2; i >= 0; --i) {
133     x[i] = -alpha[i] * x[i + 1] + betta[i];
134   }
135
136   for (int i = 1; i <= n; ++i) {
137     printf(
138         "ih = %4.2lf | y_i = %9.6lf | u(ih) = %8.6lf | |y_i - u(ih)| = "
139         "%8.6lf\n",
140         i * h, x[i - 1], u_x(i * h), abs(x[i - 1] - u_x(i * h)));
141   }
142 }
143
144 double calculate_new_x(int i, vector<double> &x, int n,
145                        vector<vector<double>> &A, vector<double> &b) {
146   double h = 1.0 / n;
147   double sum = 0.0;
148
149   if (i > 0) {
150     for (int j = 0; j <= i - 1; ++j) {
151       sum += A[i][j] * x[j];
152     }
153   }
154
155   if (i < n - 1) {
156     for (int j = i + 1; j < n + 1; ++j) {
157       sum += A[i][j] * x[j];
158     }
159   }
160
161   return (b[i] - sum) * (1.0 / A[i][i]);
162 }
163
164 int Seidel_method(int n, vector<double> &x, vector<vector<double>> &A,
165                   vector<double> &b) {
166   double h = 1.0 / n;
167   int k = 0;
168   vector<double> new_x(n, 0.);
169   double error = 1.0;
```

```
170
171   while (error > 1.0 / pow(n, 3)) {
172     for (int i = 0; i < n; ++i) {
173       new_x[i] = calculate_new_x(i, new_x, n, A, b);
174     }
175
176     error = calculate_error_dec(new_x, A, b);
177
178     x = new_x;
179
180     k++;
181   }
182
183   return k;
184 }
185 double calculate_new_x_relax(int i, vector<double> &x, int n,
186                              vector<vector<double>> &A, double omega,
187                              vector<double> &b) {
188   double sum = 0.0;
189
190   if (i > 0) {
191     for (int j = 0; j <= i - 1; ++j) {
192       sum += A[i][j] * x[j];
193     }
194   }
195
196   if (i < n - 1) {
197     for (int j = i + 1; j < n + 1; ++j) {
198       sum += A[i][j] * x[j];
199     }
200   }
201
202   double new_x = (b[i] - sum) * (1.0 / A[i][i]);
203   return x[i] + omega * (new_x - x[i]);
204 }
205
206 int relax_top(int n, vector<double> &x, vector<vector<double>> &A,
207                  vector<double> &b) {
208   double h = 1.0 / n;
209   double omega = 1.9;
210     double error = 1.0;
211     int k = 0;
212     vector<double> new_x(n, 0.);
```

```
213
214     while (error > 1.0 / pow(n, 3)) {
215       for (int i = 0; i < n; ++i) {
216         new_x[i] = calculate_new_x_relax(i, new_x, n, A, omega, b);
217       }
218
219       error = calculate_error_dec(new_x, A, b);
220
221       x = new_x;
222
223       k++;
224     }
225
226     return k;
227 }
228 double calculate_tau(vector<double> &r, vector<double> &Ar) {
229   double a = 0.;
230   double b = 0.;
231   for (size_t i = 0; i < r.size(); ++i) {
232     a += r[i] * r[i];
233     b += Ar[i] * r[i];
234   }
235   if (abs(b) < 1e-10) {
236     return 0.0;
237   }
238   return a * (1.0 / b);
239 }
240
241 double calculate_new_x_spusk(int i, vector<double> &x, int n,
242                              vector<vector<double>> &A, vector<double> &b
                                    ) {
243   vector<double> r = calculate_r(A, b, x);
244   vector<double> Ar = calculate_mtx_vec_mult(A, r);
245
246   return x[i] - calculate_tau(r, Ar) * r[i];
247 }
248
249 int spusk(int n, vector<double> &x, vector<vector<double>> &A,
250           vector<double> &b) {
251   double h = 1.0 / n;
252   int k = 1;
253   vector<double> new_x(n, 0.);
254   double error = 1.0;
```

```
255
256   while (error >= 1.0 / pow(n, 3)) {
257     for (int i = 0; i < n; ++i) {
258       new_x[i] = calculate_new_x_spusk(i, new_x, n, A, b);
259     }
260
261     x = new_x;
262
263     error = calculate_error_dec(new_x, A, b);
264
265     k++;
266   }
267
268   return k;
269 }
270
271 void m_print(int k, vector<double> &y_i, vector<double> &y_ik) {
272   for (int i = 0; i < y_i.size() - 1; ++i) {
273     printf(
274         "ih = %3d | y_i = %9.6lf | y_ik = %9.6lf | |y_i - y_ik| = %9.6lf
              | k = "
275         "%d\n",
276         i, y_i[i], y_ik[i], abs(y_i[i] - y_ik[i]), k);
277   }
278 }
```

```
1  #include "header.hpp"
2
3  using namespace std;
4
5  int main() {
6    int n = 50.0;
7
8    std::vector<double> y_i_p(n + 1), y_i_s(n + 1), y_i_r(n + 1), y_i_sp(n
       + 1),
9        b(n);
10   vector<vector<double>> A = create_matrix(n);
11   b = func_vec(n);
12   printf("Progonka\n");
13   progonka_method(A, y_i_p, n, b);
14   int k_s = Seidel_method(n, y_i_s, A, b);
15   int k_r = relax_top(n, y_i_r, A, b);
16   int k_dec = spusk(n, y_i_sp, A, b);
```

```
17    printf("Zeidel\n");
18    m_print(k_s, y_i_p, y_i_s);
19    printf("High Relaxation\n");
20    m_print(k_r, y_i_p, y_i_r);
21    printf("Spusk\n");
22    m_print(k_dec, y_i_p, y_i_sp);
23    system("pause");
24 }
```