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
студентки 219 учебной группы факультета ВМК МГУ

Чернобай Анны Александровны

Москва
2020

Оглавление

Глава 1

Решение СЛАУ методом Гаусса и методом Гаусса с выбором главного элемента

Постановка задачи

Необходимо написать программу, решающую заданную систему линейных алгебраический уравнений $Ax = y$, с невырожденной матрицей A методом Гаусса и методом Гаусса с выбором главного элемента. Размеры матрицы n x n, n – параметр задачи, задаваемый пользователем.

Матрица задается следующими способами:

1. Матрица и ее правая часть y задаются во входном файле программы или на стандартном потоке ввода.
2. Элементы матрицы вычисляются по формуле.

Задачи:

1. Решить заданную СЛАУ методом Гаусса и модифицированным методом Гаусса.
2. Вычислить определитель матрицы
3. Вычислить обратную матрицу
4. Определить число обусловленности
5. Исследовать вопрос вычислительной устойчивости метода Гаусса

Метод и алгоритм решения

Рассмотрим систему линейных алгебраических уравнений $Ax = f$, где A - матрица $= (a_{ij})$.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = f_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = f_2 \\ ... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = f_n \end{cases} \tag{1.1}$$

1. Реализация метода Гаусса

   Метод удобно разделить на два основных этапа:

   - Прямой ход: система приводится к треугольному виду.
   - Обратный ход: последовательное отыскание неизвестных $x_1, x_2, ..., x_n$.

   Прямой ход:
   На данном этапе необходимо привести исходную систему линейных уравнений к эквивалентой системе, имеющей верхнюю треугольную матрицу. Для этого рассмотрим коэффицент a_{11}. Если $a_{11} = 0$, то найдем первый коэффицент такой, что $a_{i1} \neq 0$, и затем поменяем 1-ое и i-ое уравнения местами. Такое i будет существовать в силу невырожденности матрицы A.
   Разделим все члены 1-го уравнения на a_{11}. Затем вычтем каждой i-ой строки 1-ое уравнение, умноженное на a_{i1} соответственно:

$$\begin{cases} x_1 + \frac{a_{12}}{a_{11}}x_2 + ... + \frac{a_{1n}}{a_{11}}x_n = f_1 \\ 0 + (a_{22} - \frac{a_{12}a_{21}}{a_{11}})x_2 + ... + (a_{2n} - \frac{a_{1n}a_{21}}{a_{11}})x_n = f_2 - \frac{a_{21}}{a_{11}}f_1 \\ ... \\ 0 + (a_{n2} - \frac{a_{12}a_{n1}}{a_{11}})x_2 + ... + (a_{nn} - \frac{a_{1n}a_{n1}}{a_{11}})x_n = f_n - \frac{a_{n1}}{a_{11}}f_1 \end{cases} \tag{1.2}$$

   После данных преобразований мысленно уберем 1-ый столбец матрицы и проведем аналогичные преобразования со 2-ым столбцом и т.д. После этих преобразований получим следующую систему:

$$\begin{cases} a'_{11}x_1 + a'_{12}x_2 + ... + a'_{1n}x_n = f'_1 \\ 0 + x_2 + ... + a'_{2n}x_n = f'_2 \\ ... \\ 0 + 0 + ... + x_n = f'_n \end{cases} \tag{1.3}$$

   Обратный ход: Данный этап начнем с n-го уравнения и закончим 1-ым. Из n-го уравнения получим x_n. Подставим его в (n-1)-ое уравнение и получим x_{n-1}. Проведем аналогичные преобразования с оставшимися уравнениями, в процессе получим решение системы.

2. Метод Гаусса с выбором главного элемента
   Модифицированный метод Гаусса отличается от рассмотренного в 1 пункте тем, что на i-ом шаге прямого хода выбираем наибольший по модулю элемент (он всегда будет ненулевым в силу невырожденности матрицы) i-ой строки $a_{ijmax} = \max\limits_{i \leq j \leq n} |a_{ij}|$ и меняем в системе местами i-ый столбец и столбец с номером j_{max}. Все остальные шаги выполняются аналогично.

3. Определитель матрицы
   Для вычисления определителя матрицы заметим, что определитель треугольной матрицы равен произведению ее диагональных элементов. Для приведения матрицы к верхнетреугольному виду воспользуемся алгоритмом, описанным в прямом ходе метода Гаусса и будем учитывать, что при делении строки на элемент матрицы, определитель матрицы умножается на этот элемент и что перестановка 2-х строк матрицы меняет знак определителя.

$$\Delta = (-1)^k a_{11} a'_{22} ... a'_{nn} \tag{1.4}$$

4. Обратная матрица
   Обратную матрицу вычислим следующим образом:если с единичной матрицей I провести элементарные преобразования, которыми невырожденная квадратная матрица приводится к I, то получится обратная матрица A^{-1}.
   Приведение матрицы к единичной заключается в приведении ее сначала к верхнетреугольной, а затем к нижнетреугольной методом из п.1 Гаусса (таким образом главная диагональ будет нормирована).
   Все преобразования будем проводить с расширенной матрицей $[A|I]$.

5. Число обусловленности
   Будем считать норму матрицы следующим образом: $||A|| = \max\limits_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. Число обусловленности $M_A = ||A||||A^{-1}||$.

Структура программы и спецификация функций

Программа состоит из одного модуля task1gauss.c. В данном модуле реализовано заполнение матрицы размера $n \times n$ и функции, описанные ниже:

Спецификация функций:

- `double norm(double **koef, int n);`

  Функция считает норму матрицы размерности n, элементы которой хранятся по адресу $koef$, по формуле $||A|| = \max\limits_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$.

- `double condition_number(double **a, double **a_reverse, int n);`

  Функция считает число обусловленности матрицы размерности n, элементы которой хранятся по адресу a,по формуле $M_A = ||A||||A^{-1}||$. Внутри функции нормы считаются с помощью вызова функции $norm$.

- `double determinant(int n, double **koef);`

  Функция считает определитель матрицы размерности n, элементы которой хранятся по адресу $koef$.

- `void gauss(int n, double **koef, double *ans);`

  Функция решает систему из n линейных алгебраических уравнений методом Гаусса, матрица системы записана по адресу $koef$. Решение системы записывается по адресу ans.

- `void reverse_matrix(int n, double **koef, double **reverse);`

  Функция считает обратную матрицу для матрицы размерности n, элементы которой хранятся по адресу $koef$, методом Гаусса-Жордана. Полученная обратная матрица записывается по адресу $reverse$.

- `void gauss_with_main(int n, double **koef, double *ans, int *x);`

  Функция решает систему из n линейных алгебраических уравнений методом Гаусса с выбором главного эелемента, матрица системы записана по адресу $koef$. Решение системы записывается по адресу ans.

- `int main(void);`

  Основная функция программы. В ней осуществляется задание системы линейных алгебраических уравнений $Ax = f$ одним из 3-х способов: из файла(1), из консоли(2) или с помощью формул(3). Далее вычисляется определитель матрицы A с помошью функции $determinant$. Если определитель матрицы равен нулю, программа выводит пользователю определитель матрицы и затем завершает свою работу. Если определитель не равен нулю, то далее будут вызваны функции $reverse\_matrix$ и $conditional\_number$. Далее пользователью будет предложено выбрать метод (Гаусс(1) или Гаусс с выбором главного элемента(2)), которым необходимо решить систему и будет вызвана соотвествующая функция. После вывода решения системы программа завершит свою работу.

Программа на СИ

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <inttypes.h>
#include <limits.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double norm(double **koef, int n) {
    double norm = 0;
    double sum;
    for (int i = 0; i < n; i++) {
        sum = 0;
        for (int j = 0; j < n; j++) {
            sum += fabs(koef[i][j]);
        }
        if (sum > norm) {
            norm = sum;
        }
    }
    return norm;
}

double condition_number(double **a, double **a_reverse, int n) {
    return norm(a, n) * norm(a_reverse, n);
}

double determinant(int n, double **koef) {
    double eps = 0.000000001, exc, det = 1;
    int flag;
    for (int i = 0; i < n; i++) {
        if (fabs(koef[i][i]) <= eps) {
            flag = 0;
            for (int j = i + 1; j < n; j++) {
                if (fabs(koef[j][i]) > eps) {
                    flag = 1;
                    det = det * (-1);
                    for (int l = 0; l < n; l++) {
                        exc = koef[i][l];
                        koef[i][l] = koef[j][l];
                        koef[j][l] = exc;
```

```
                }
                break;
            }
        }
        if (flag == 0) {
            return 0;
        }
    }
    exc = koef[i][i];
    det *= exc;
    for (int j = i; j < n; j++) {
        koef[i][j] = koef[i][j] / exc;
    }
    for (int j = i + 1; j < n; j++) {
        exc = koef[j][i];
        for(int l = i; l < n; l++ ) {
            koef[j][l] -= exc * koef[i][l];
        }
    }
}
return det;
}

void gauss(int n, double **koef, double *ans) {
    double eps = 0.000001, exc;
    for (int i = 0; i < n; i++) {
        if (fabs(koef[i][i]) <= eps) {
            for (int j = i + 1; j < n; j++) {
                if (fabs(koef[j][i]) > eps) {
                    for (int l = 0; l <= n; l++) {
                        exc = koef[i][l];
                        koef[i][l] = koef[j][l];
                        koef[j][l] = exc;
                    }
                    break;
                }
            }
        }
        exc = koef[i][i];
        for (int j = i; j <= n; j++) {
            koef[i][j] = koef[i][j] / exc;
        }
        for (int j = i + 1; j < n; j++) {
            exc = koef[j][i];
            for(int l = i; l <= n; l++ ) {
                koef[j][l] -= exc * koef[i][l];
```

```
            }
        }
    }
    for (int i = 0; i < n; i++) {
        ans[i] = koef[i][n];
    }
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
             ans[i] -= koef[i][j] * ans[j];
        }
        ans[i] /= koef[i][i];
    }
}

void reverse_matrix(int n, double **koef, double **reverse) {
    double eps = 0.000001, exc;
    for (int i = 0; i < n; i++) {
        if (fabs(koef[i][i]) <= eps) {
            for (int j = i; j < n; j++) {
                if (fabs(koef[j][i]) > eps) {
                    for (int l = 0; l < n; l++) {
                        exc = koef[i][l];
                        koef[i][l] = koef[j][l];
                        koef[j][l] = exc;
                        exc = reverse[i][l];
                        reverse[i][l] = reverse[j][l];
                        reverse[j][l] = exc;
                    }
                    break;
                }
            }
        }
        exc = koef[i][i];
        for (int j = 0; j < n; j++) {
            koef[i][j] = koef[i][j] / exc;
            reverse[i][j] = reverse[i][j] / exc;
        }
        for (int j = i + 1; j < n; j++) {
            exc = koef[j][i];
            for(int l = 0; l < n; l++ ) {
                koef[j][l] -= exc * koef[i][l];
                reverse[j][l] -= exc * reverse[i][l];
            }
        }
    }
    for (int i = n - 1; i >= 0; i--) {
```

```
        for (int j = 0; j < i; j++) {
            exc = koef[j][i];
            for (int l = 0; l < n; l++) {
                reverse[j][l] -= exc * reverse[i][l];
            }
        }
    }
}

void gauss_with_main(int n, double **koef, double *ans, int *x) {
    double eps = 0.000001, exc, max;
    int i_max;
    for (int i = 0; i < n; i++) {
        max = fabs(koef[i][i]);
        i_max = i;
        for (int j = i; j < n; j++) {
            if (fabs(koef[i][j]) > max) {
                max = fabs(koef[i][j]);
                i_max = j;
            }
        }
        for (int j = 0; j < n; j++) {
            exc = koef[j][i];
            koef[j][i] = koef[j][i_max];
            koef[j][i_max] = exc;
        }
        exc = x[i];
        x[i] = x[i_max];
        x[i_max] = exc;
        exc = koef[i][i];
        for (int j = i; j <= n; j++) {
            koef[i][j] = koef[i][j] / exc;
        }
        for (int j = i + 1; j < n; j++) {
            exc = koef[j][i];
            for(int l = i; l <= n; l++ ) {
                koef[j][l] -= exc * koef[i][l];
            }
        }
    }
    for (int i = 0; i < n; i++) {
        ans[i] = koef[i][n];
    }
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
             ans[i] -= koef[i][j] * ans[j];
```

```
        }
        ans[i] /= koef[i][i];
    }
}

int main(void) {
     int input_type, n;
     double eps = 0.00000000001;
     printf("Enter n (size of matrix) ");
     scanf("%d", &n);
     double **koef, **reverse_koef, **check_deter;
     koef = malloc(n * sizeof(double *));
     reverse_koef = malloc(n * sizeof(double *));
     check_deter = malloc(n * sizeof(double *));
     if (koef == NULL || reverse_koef == NULL || check_deter == NULL) {
         printf("UNABLE TO RESERVE MEMORY");
         exit(1);
     }
     for (int i = 0 ; i < n; i++) {
         koef[i] = calloc(n + 1, sizeof(double));
         reverse_koef[i] = calloc(n , sizeof(double));
         check_deter[i] = calloc(n, sizeof(double));
         if (koef[i] == NULL || reverse_koef[i] == NULL || check_deter[i] == NULL) {
              printf("UNABLE TO RESERVE MEMORY");
              exit(1);
         }
     }
     printf("Choose the type of input: \n");
     printf("Press 1 to choose input from file\n");
     printf("Press 2 to choose input from console\n");
     printf("Press 3 to choose input based on formula\n");
     scanf("%d", &input_type);
     if (input_type == 1) {
         printf("Enter file:");
         char filename[PATH_MAX];
         scanf("%s", filename);
         FILE *fd = fopen(filename, "r");
         if (fd == NULL) {
             printf("UNABLE TO OPEN THE FILE");
             exit(2);
         }
         for (int i = 0; i < n; i++) {
             for (int j = 0; j <= n; j++) {
                 fscanf(fd, "%lf", &koef[i][j]);
                 if (j < n) {
                     reverse_koef[i][j] = koef[i][j];
```

```
                check_deter[i][j] = koef[i][j];
            }
        }
    }
}
if (input_type == 2) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            scanf("%lf", &koef[i][j]);
        }
    }
   for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            reverse_koef[i][j] = koef[i][j];
            check_deter[i][j] = koef[i][j];
        }
    }
}
if (input_type == 3) {
    double q, x_form;
    q = 1.001 - 12 * 0.001;
    printf("Enter x: ");
    scanf("%lf", &x_form);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                koef[i][j] = pow(q - 1, i + j + 2);
            }
            else {
                koef[i][j] = pow(q, i + j + 2) + 0.1 * (j - i);
            }
            reverse_koef[i][j] = koef[i][j];
            check_deter[i][j] = koef[i][j];
        }
    }
    for (int i = 0; i < n; i++) {
        koef[i][n] = x_form * exp(x_form / (i + 1)) * cos(x_form / (i + 1));
    }
}
double det_a = determinant(n, check_deter);
if (fabs (det_a) > eps) {
    double **reverse = malloc(n * sizeof(double *));
    if (reverse == NULL) {
        printf("UNABLE TO RESERVE MEMORY");
        exit(1);
    }
```

```
for (int i = 0 ; i < n; i++) {
    reverse[i] = calloc(n, sizeof(double));
    if (reverse[i] == NULL) {
        printf("UNABLE TO RESERVE MEMORY");
        exit(1);
    }
    reverse[i][i] = 1;
}
double *ans = calloc(n, sizeof(double));
if (ans == NULL) {
    exit(1);
}
printf("|A| = %.2lf\n", det_a);
reverse_matrix(n, reverse_koef, reverse);
printf("REVERSE MATRIX:\n");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        printf("%.2lf ", reverse[i][j]);
    }
    printf("\n");
}
printf("CONDITION_NUMBER: %.4lf \n", condition_number(koef, reverse, n));
printf("Choose method: \n");
printf("Press 1 to choose standart gauss method\n");
printf("Press 2 to choose modified gauss method\n");
scanf("%d", &input_type);
if (input_type == 1) {
    gauss(n, koef, ans);
    for (int i = 0; i < n; i++) {
        printf("%.4lf ", ans[i]);
    }
}
else {
    int *x = malloc(n * sizeof(int));
    if (x == NULL) {
        exit(1);
    }
    for (int i = 0; i < n; i++) {
        x[i] = i;
    }
    gauss_with_main(n, koef, ans, x);
    for (int i = 0; i < n; i++) {
        printf("%.4lf ", ans[x[i]]);
    }
    printf("\n");
    free(x);
```

```
        }
        for (int i = 0; i < n; i++) {
            free(reverse[i]);
        }
        free(ans);
        free(reverse);

    }
    else {
        printf("|A| = 0");
    }
    for (int i = 0; i < n; i++) {
        free(koef[i]);
        free(reverse_koef[i]);
        free(check_deter[i]);
    }
    free(check_deter);
    free(reverse_koef);
    free(koef);
    return 0;
}
```

Тестирование и результаты работы программы

Тестирование программы проводилось в режиме 1 на 3-х тестах, взятых из приложения 1 - 7:

СЛАУ	Ожидаемые результаты	Результаты работы программы
$2x_1 - x_2 - 6x_3 + 3x_4 = -1$ $2x_1 - 4x_2 + 2x_3 - 15x_4 = -32$ $2x_1 - 2x_2 - 4x_3 + 9x_4 = 5$ $2x_1 - x_2 + 2x_3 - 6x_4 = -8$	$M_A = 51\frac{1}{3}$ $\|A\| = -252$ $X = (-3, 0, -\frac{1}{2}, \frac{2}{3})$ Inverse matrix $\begin{pmatrix} -\frac{3}{14} & \frac{8}{21} & -\frac{1}{14} & -\frac{7}{6} \\ -\frac{1}{14} & \frac{5}{21} & -\frac{5}{14} & -\frac{7}{6} \\ -\frac{2}{7} & \frac{5}{42} & \frac{1}{14} & -\frac{1}{3} \\ -\frac{5}{42} & \frac{4}{63} & \frac{1}{14} & -\frac{5}{18} \end{pmatrix}$	$M_A = 51.3333$ $\|A\| = -252.00$ $X = (-3.00, 0.00, -0.50, 0.67)$ Inverse matrix $\begin{pmatrix} -0.21 & 0.38 & -0.07 & -1.17 \\ -0.07 & 0.24 & -0.36 & -1.17 \\ -0.29 & 0.12 & 0.07 & -0.33 \\ -0.12 & 0.06 & 0.07 & -0.28 \end{pmatrix}$
$x_1 - 2x_2 + x_3 + x_4 = 0$ $2x_1 + x_2 - x_3 - x_4 = 0$ $3x_1 - x_2 - 2x_3 + x_4 = 0$ $5x_1 + 2x_2 - x_3 + 9x_4 = -10$	$M_A = 23.0508$ $\|A\| = -118$ $X = (-\frac{15}{59}, -\frac{45}{59}, -\frac{25}{59}, -\frac{50}{59})$ Inverse matrix $\begin{pmatrix} \frac{33}{118} & \frac{25}{59} & -\frac{5}{59} & \frac{3}{118} \\ -\frac{19}{118} & \frac{16}{59} & -\frac{15}{59} & \frac{9}{118} \\ \frac{55}{118} & \frac{22}{59} & -\frac{28}{59} & \frac{5}{118} \\ -\frac{4}{59} & -\frac{15}{59} & \frac{3}{59} & \frac{5}{59} \end{pmatrix}$	$M_A = 23.0508$ $\|A\| = -118.00$ $X = (-0.25, -0.76, -0.42, -0.85)$ Inverse matrix $\begin{pmatrix} 0.28 & 0.42 & -0.08 & 0.03 \\ -0.16 & 0.27 & -0.25 & 0.08 \\ 0.47 & 0.37 & -0.47 & 0.04 \\ -0.07 & -0.25 & 0.05 & 0.08 \end{pmatrix}$
$8x_1 + 6x_2 + 5x_3 + 2x_4 = 21$ $3x_1 + 3x_2 + 2x_3 + x_4 = 10$ $4x_1 + 2x_2 + 3x_3 + x_4 = 8$ $7x_1 + 4x_2 + 5x_3 + 2x_4 = 18$	$M_A = 210$ $\|A\| = 1$ $X = (3, 0, -5, 11)$ Inverse matrix $\begin{pmatrix} 1 & -2 & -2 & 1 \\ 0 & 1 & 1 & -1 \\ -1 & 2 & 4 & -2 \\ -1 & 0 & -5 & 4 \end{pmatrix}$	$M_A = 210.0000$ $\|A\| = 1.00$ $X = (3.00, 0.00, -5.00, 11.00)$ Inverse matrix $\begin{pmatrix} 1.00 & -2.00 & -2.00 & 1.00 \\ -0.00 & 1.00 & 1.00 & -1.00 \\ -1.00 & 2.00 & 4.00 & -2.00 \\ -1.00 & 0.00 & -5.00 & 4.00 \end{pmatrix}$

Так же тестирование программы проводилось на тестах, построенных по формулам, взятых из приложения 2 п.2-6:

$$A_{ij} = \begin{cases} q_M^{(i+j)} + 0.1(j-i), j \neq i \\ (q_M - 1)^{(i+j)}, j = i \end{cases} \quad \text{где } q_M = 1.001 - 2 * M * 10^{-3}$$

$f_i = x * e^{\frac{x}{i}} * cos(\frac{x}{i})$

Результаты работы программы

СЛАУ	Результаты работы программы
n = 3 x = 1	$\|A\| = 1.69$ $M_A = 3.9140$ $X = (0.9379, 0.7198, 0.6054)$ $\begin{pmatrix} -0.52 & 0.58 & 0.66 \\ 0.47 & -0.52 & 0.59 \\ 0.43 & 0.48 & -0.55 \end{pmatrix}$
n = 4 x = 2	$\|A\| = -2.23$ $M_A = 7.1261$ $X = (8.3493, -1.3225, -1.8515, -2.0844)$ $\begin{pmatrix} -0.71 & 0.38 & 0.45 & 0.52 \\ 0.30 & -0.69 & 0.40 & 0.46 \\ 0.29 & 0.32 & -0.71 & 0.40 \\ 0.28 & 0.29 & 0.31 & -0.76 \end{pmatrix}$
n = 2 x = 0	$\|A\| = -0.93$ $M_A = 1.2307$ $X = (0.0000 - 0.0000)$ $\begin{pmatrix} -0.00 & 1.15 \\ 0.94 & -0.00 \end{pmatrix}$

Выводы

В данной работе необходимо было изучить и реализовать метод Гаусса решения СЛАУ с невырожденной матрицей. Сам метод прост в реализации, однако решения СЛАУ, полученные с его помощью достаточно часто не являются точными. Неизбежны как ошибки входных данных, так и ошибки округления - каждое вычисление будет неизбежно иметь неточность из-за вычислительных возможностей компьютера.

Также, анализируя результаты работы программы на тестах, можно сделать вывод о том, что на устойчивость метода влияла сама матрица A. Так, на матрицах с большим числом обусловленности (как минимум, больше 210, если отталкиваться от тестировки) данный метод очень неустойчив. В качестве примера возьмем СЛАУ:

$$\begin{cases} x_1 + x_2 = 2 \\ x_1 + 1.0001x_2 = 2.0001 \end{cases} \tag{1.5}$$

$$\begin{cases} x_1 + x_2 = 2 \\ x_1 + 1.0001x_2 = 2.0002 \end{cases} \tag{1.6}$$

Число обусловленности и в 1-ой, и во 2-ой системе $M_A = 40004.00001$. Результатом работы программы для 1-ой СЛАУ будет вектор $(1.00, 1.00)$, а для 2-ой СЛАУ - $(0.00, 2.00)$. Легко заметить, как незначительное изменение правой части СЛАУ повлияло на результат работы программы.

Таким образом, чем ниже обусловленность матрицы тем устойчивее будет метод на данной СЛАУ.

Глава 2

Итерационный метод решения СЛАУ. Метод верхней релаксации

Постановка задачи

Необходимо написать программу решающую заданную систему линейных алгебраический уравнений $Ax = y$, с невырожденной матрицей A, использующую численный алгоритм итерационного метода верхней релаксации. Размеры матрицы n x n, n – параметр задачи, задаваемый пользователем. Матрица A задается способами, аналогичными описанным в предыдущей задаче на метод Гаусса. Задачи:

1. Решить заданную СЛАУ методом верхней релаксации
2. Разработать критерий остановки итерационного процесса, гарантирующий вычисление приближенного уравнения с заданной точностью
3. Изучить скорость сходимости итераций к точному решению задачи

Метод и алгоритм решения

Рассмотрим систему линейных алгебраических уравнений $Ax = f$, где $A = (a_{ij})$ - невырожденная матрица.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = f_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = f_2 \\ ... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = f_n \end{cases} \tag{2.1}$$

1. Решение СЛАУ методом верхней релаксации
   Для реализации метода верхней релаксации нужно, чтобы матрица A была симметрической. Воспользуемся преобразованием, приводящим систему к эквивалентной:

$$A^T Ax = A^T f \tag{2.2}$$

   Введем следующие обозначения:
   $A = T_H + D + T_B$
   T_H - нижняя треугольная матрица размера $n \times n$, где главная диагональ нулевая, а поддиагональные элементы совпадают с соответствующими элементами матрицы A
   T_B - верхняя треугольная матрица размера $n \times n$, где главная диагональ нулевая, а наддиагональные элементы совпадают с соответствующими элементами матрицы A
   D - диагональная матрица, где главная диагональ совпадает с главной диагональю матрицы A
   ω - итерационный параметр, принимающий значения в интервале $(0; 2)$
   $x^{(k)}$ - приближение, полученное на k-ой итерации
   Используя эти обозначения итерационный метод верхней релаксации можно записать в следующем виде:

$$(D + \omega T_H)\frac{(x^{k+1} - x^k)}{\omega} + Ax^k = f \tag{2.3}$$

$$x_i^{k+1} = x_i k + \frac{\omega}{a_{ii}}(f_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^{n} a_{ij}x_j^k) \tag{2.4}$$

2. Критерий остановки итерационного процесса
   Пусть ε - точность, с которой необходимо вычислить приближенное значение. Тогда критерием остановки будет следующее неравенство $||x^k - x|| < ||A^{-1}||||\psi^k|| = \varepsilon$, где $\psi^k = Ax^k - f$.

Структура программы и спецификация функций

Программа состоит из одного модуля task1relaxation.c. В данном модуле реализовано заполнение матрицы размера $n \times n$ и функции, описанные ниже:

Спецификация функций:

- `void reverse_matrix(int n, double **koef, double **reverse)`

  Функция считает обратную матрицу для матрицы размерности n, элементы которой хранятся по адресу $koef$, методом Гаусса-Жордана. Полученная обратная матрица записывается по адресу $reverse$.

- `double determinant(int n, double **koef)`

  Функция считает определитель матрицы размерности n, элементы которой хранятся по адресу $koef$.

- `void multiplication(int m, int n, int k, double **a, double **b, double **c)`

  Функция реализовывает перемножение матриц A и B, элементы которых хранятся по адресам a и b соотвественно. Результат перемножения записывается в матрицу C, элементы которой хранятся по адресу c.

- `double matrix_norm(double **koef, int n)`

  Функция считает норму матрицы размерности n, элементы которой хранятся по адресу $koef$, по формуле $||A|| = \max\limits_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$.

- `int relaxation(int n, double **ans, double **matrix, double **f, double w)`

  Функция вычисляет приближенное решение СЛАУ итерационным методом верхней релаксации с параметром ω.

- `int main(void)`

  Основная функция программы. В ней осуществляется задание системы линейных алгебраических уравнений $Ax = f$ одним из 3-х способов: из файла(1), из консоли(2) или с помощью формул(3). Далее вычисляется определитель матрицы A с помошью функции $determinant$. Если определитель матрицы равен нулю, программа выводит пользователю определитель матрицы и затем завершает свою работу. Если определитель не равен нулю, то далее заданная СЛАУ будет передана в функцию $relaxation$ и будет выведено на печать приближенное решение СЛАУ итерационным методом верхней релаксации с параметром ω.

Программа на СИ

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <inttypes.h>
#include <limits.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void reverse_matrix(int n, double **koef, double **reverse) {
    double eps = 0.000001, exc;
    for (int i = 0; i < n; i++) {
        if (fabs(koef[i][i]) <= eps) {
            for (int j = i; j < n; j++) {
                if (fabs(koef[j][i]) > eps) {
                    for (int l = 0; l < n; l++) {
                        exc = koef[i][l];
                        koef[i][l] = koef[j][l];
                        koef[j][l] = exc;
                        exc = reverse[i][l];
                        reverse[i][l] = reverse[j][l];
                        reverse[j][l] = exc;
                    }
                    break;
                }
            }
        }
        exc = koef[i][i];
        for (int j = 0; j < n; j++) {
            koef[i][j] = koef[i][j] / exc;
            reverse[i][j] = reverse[i][j] / exc;
        }
        for (int j = i + 1; j < n; j++) {
            exc = koef[j][i];
            for(int l = 0; l < n; l++ ) {
                koef[j][l] -= exc * koef[i][l];
                reverse[j][l] -= exc * reverse[i][l];
            }
        }
    }
    for (int i = n - 1; i >= 0; i--) {
        for (int j = 0; j < i; j++) {
```

```
            exc = koef[j][i];
            for (int l = 0; l < n; l++) {
                reverse[j][l] -= exc * reverse[i][l];
            }
        }
    }
}

double determinant(int n, double **koef) {
    double eps = 0.000000001, exc, det = 1;
    int flag;
    for (int i = 0; i < n; i++) {
        if (fabs(koef[i][i]) <= eps) {
            flag = 0;
            for (int j = i + 1; j < n; j++) {
                if (fabs(koef[j][i]) > eps) {
                    flag = 1;
                    det = det * (-1);
                    for (int l = 0; l < n; l++) {
                        exc = koef[i][l];
                        koef[i][l] = koef[j][l];
                        koef[j][l] = exc;
                    }
                    break;
                }
            }
            if (flag == 0) {
                return 0;
            }
        }
        exc = koef[i][i];
        det *= exc;
        for (int j = i; j < n; j++) {
            koef[i][j] = koef[i][j] / exc;
        }
        for (int j = i + 1; j < n; j++) {
            exc = koef[j][i];
            for(int l = i; l < n; l++ ) {
                koef[j][l] -= exc * koef[i][l];
            }
        }
    }
    return det;
}

void multiplication(int m, int n, int k, double **a, double **b, double **c) {
```

```
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            c[i][j] = 0;
            for (int l = 0; l < n; l++) {
                c[i][j] += a[i][l] * b[l][j];
            }
        }
    }
}

double matrix_norm(double **koef, int n) {
    double norm = 0;
    double sum;
    for (int i = 0; i < n; i++) {
        sum = 0;
        for (int j = 0; j < n; j++) {
            sum += fabs(koef[i][j]);
        }
        if (sum > norm) {
            norm = sum;
        }
    }
    return norm;
}

int relaxation(int n, double **ans, double **matrix, double **f, double w) {
    double *x_prev;
    double **x_tmp = malloc(n * sizeof(double *));
    double **reverse, **tmpreverse;
    reverse = malloc(n * sizeof( double *));
    tmpreverse = malloc(n * sizeof(double *));
    if (reverse == NULL || tmpreverse == NULL || x_tmp == NULL) {
        exit(1);
    }
    for (int i = 0; i < n; i++){
        tmpreverse[i] = calloc(n, sizeof(double));
        reverse[i] = calloc(n, sizeof(double));
        x_tmp[i] = calloc(1, sizeof(double));
        if (reverse[i] == NULL || tmpreverse[i] == NULL || x_tmp[i] == NULL) {
            exit(1);
        }
        for (int j = 0; j < n; j++) {
            tmpreverse[i][j] = matrix[i][j];
        }
        reverse[i][i] = 1;
    }
```

```
reverse_matrix(n, tmpreverse, reverse);
double norm_reverse = matrix_norm(reverse, n);
x_prev = calloc(n, sizeof(double));
if (x_prev == NULL) {
    exit(1);
}
for (int i = 0; i < n; i++) {
    ans[i][0] = 0;
}
double eps = 0.001, sum = 0;
int count_iterations = 0;
double norm_psi = 0;
do {
    count_iterations++;
    for (int i = 0; i < n; i++) {
        x_prev[i] = ans[i][0];
    }
    for (int i = 0; i < n; i++) {
        sum = 0;
        for (int j = 0; j < i; j++) {
            sum += (matrix[i][j] * ans[j][0]);
        }
        for (int j = i; j < n; j++) {
            sum += (matrix[i][j] * x_prev[j]);
        }
        ans[i][0] = w * (f[i][0] - sum)/ matrix[i][i] + x_prev[i];
    }
    norm_psi = 0;
    multiplication(n, n, 1, matrix, ans, x_tmp);
    for (int i = 0; i < n; i++) {
        x_tmp[i][0] -= f[i][0];
        if (fabs(x_tmp[i][0]) > norm_psi) {
            norm_psi = fabs(x_tmp[i][0]);
        }
    }
} while (fabs(norm_reverse * norm_psi) > eps);
for (int i = 0; i < n; i++) {
    free(tmpreverse[i]);
    free(reverse[i]);
    free(x_tmp[i]);
}
free(tmpreverse);
free(reverse);
free(x_tmp);
free(x_prev);
return count_iterations;
```

```
}

int main(void) {
     int input_type, n;
     double w;
     printf("Enter n (size of matrix) and w (omega): ");
     scanf("%d %lf", &n, &w);
     double **koef, **trans_koef, **sym, **check_deter;
     koef = malloc(n * sizeof(double *));
     trans_koef = malloc(n * sizeof(double *));
     sym = malloc(n * sizeof(double *));
     check_deter = malloc(n * sizeof(double *));
     if (koef == NULL || trans_koef == NULL || sym == NULL || check_deter == NULL) {
         printf("UNABLE TO RESERVE MEMORY");
         exit(1);
     }
     for (int i = 0 ; i < n; i++) {
         koef[i] = calloc(n , sizeof(double));
         trans_koef[i] = calloc(n , sizeof(double));
         sym[i] = calloc(n , sizeof(double));
         check_deter[i] = calloc(n , sizeof(double));
         if (koef[i] == NULL || trans_koef[i] == NULL || sym[i] == NULL || check_deter[i] == 
              printf("UNABLE TO RESERVE MEMORY");
              exit(1);
         }
     }
     double **f, **tmp, **ans;
     f = malloc(n * sizeof(double *));
     tmp = malloc(n * sizeof(double *));
     ans = malloc(n * sizeof(double *));
     if (f == NULL || tmp == NULL || ans == NULL) {
         printf("UNABLE TO RESERVE MEMORY");
         exit(1);
     }
     for (int i = 0 ; i < n; i++) {
         f[i] = calloc(1 , sizeof(double));
         tmp[i] = calloc(1 , sizeof(double));
         ans[i]= calloc(1 , sizeof(double));
         if (f[i] == NULL || tmp[i] == NULL || ans[i] == NULL) {
              printf("UNABLE TO RESERVE MEMORY");
              exit(1);
         }
     }
     printf("Choose the type of input: \n");
     printf("Press 1 to choose input from file\n");
     printf("Press 2 to choose input from console\n");
```

```
printf("Press 3 to choose input based on formula\n");
scanf("%d", &input_type);
if (input_type == 1) {
    printf("Enter file:");
    char *filename;
    scanf("%s", filename);
    FILE *fd = fopen(filename, "r");
    if (fd == NULL) {
        printf("UNABLE TO OPEN THE FILE");
        exit(2);
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            if (j < n) {
                fscanf(fd, "%lf", &koef[i][j]);
                trans_koef[j][i] = koef[i][j];
                check_deter[i][j] = koef[i][j];
            }
            else {
                 fscanf(fd, "%lf", &f[i][0]);
            }
        }
    }
}
if (input_type == 2) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            if (j < n) {
                scanf( "%lf", &koef[i][j]);
                trans_koef[j][i] = koef[i][j];
                check_deter[i][j] = koef[i][j];
            }
            else {
                scanf( "%lf", &f[i][0]);
            }
        }
    }
}
if (input_type == 3) {
    double q, x;
    q = 1.001 - 12 * 0.001;
    printf("Enter x: ");
    scanf("%lf", &x);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
```

```
                koef[i][j] = pow(q - 1, i + j + 2);
            }
            else {
                koef[i][j] = pow(q, i + j + 2) + 0.1 * (j - i);
            }
            trans_koef[j][i] = koef[i][j];
            check_deter[i][j] = koef[i][j];
        }
    }
    for (int i = 0; i < n; i++) {
        f[i][0] = x * exp(x / (i + 1)) * cos(x / (i + 1));
    }
   for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%lf ", check_deter[i][j]);
        }
        printf("\n");
    }
}
double eps = 0.000000000000001;
double det = determinant(n, check_deter);
if (fabs(det) < eps) {
    printf("%lf\n", det);
    printf("Degenerate koef matrix.\n");
}
else {
    multiplication(n, n, n, trans_koef, koef, sym);
    multiplication(n, n, 1, trans_koef, f, tmp);
    int count_iterations = relaxation(n, ans, sym, tmp, w);
    for (int i = 0; i < n; i++) {
        printf("%.3lf ", ans[i][0]);
    }
    printf("\nNumber of iterations %d\n", count_iterations);
}
for (int i = 0; i < n; i++) {
    free(koef[i]);
    free(trans_koef[i]);
    free(sym[i]);
    free(check_deter[i]);
    free(tmp[i]);
    free(f[i]);
    free(ans[i]);
}
free(ans);
free(check_deter);
free(koef);
```

```
    free(trans_koef);
    free(sym);
    free(tmp);
    free(f);
    return 0;
}
```

Тестирование и результаты работы программы

Тестирование программы проводилось в режиме 1 на 3-х тестах, взятых из приложения 1 - 7:

СЛАУ	Ожидаемые результаты	Результаты работы программы
$2x_1 - x_2 - 6x_3 + 3x_4 = -1$ $2x_1 - 4x_2 + 2x_3 - 15x_4 = -32$ $2x_1 - 2x_2 - 4x_3 + 9x_4 = 5$ $2x_1 - x_2 + 2x_3 - 6x_4 = -8$	$X = (-3, 0, -\frac{1}{2}, \frac{2}{3})$	$X = (-3.000, 0.000, -0.500, 0.667)$ N = 628 $\omega = 1$ N = 387 $\omega = 1.3$ N = 178 $\omega = 1.6$ N = 366 $\omega = 1.9$
$x_1 - 2x_2 + x_3 + x_4 = 0$ $2x_1 + x_2 - x_3 - x_4 = 0$ $3x_1 - x_2 - 2x_3 + x_4 = 0$ $5x_1 + 2x_2 - x_3 + 9x_4 = -10$	$X = (-\frac{15}{59}, -\frac{45}{59}, -\frac{25}{59}, -\frac{50}{59})$	$X = (-0.254, -0.763, -0.424, -0.847)$ N = 79 $\omega = 1$ N = 42 $\omega = 1.3$ N = 34 $\omega = 1.6$ N = 155 $\omega = 1.9$
$8x_1 + 6x_2 + 5x_3 + 2x_4 = 21$ $3x_1 + 3x_2 + 2x_3 + x_4 = 10$ $4x_1 + 2x_2 + 3x_3 + x_4 = 8$ $7x_1 + 4x_2 + 5x_3 + 2x_4 = 18$	$X = (3, 0, -5, 11)$	$X = (3.000, 0.000, -5.000, 11.000)$ N = 16068 $\omega = 1$ N = 8109 $\omega = 1.3$ N = 5339 $\omega = 1.6$ N = 19591 $\omega = 1.9$

Так же тестирование программы проводилось на тестах, построенных по формулам, взятых из приложения 2 п.2-6:

$$A_{ij} = \begin{cases} q_M^{(i+j)} + 0.1(j-i), j \neq i \\ (q_M - 1)^{(i+j)}, j = i \end{cases} \quad \text{где } q_M = 1.001 - 2 * M * 10^{-3}$$

$f_i = x * e^{\frac{x}{i}} * cos(\frac{x}{i})$

Результаты работы программы

СЛАУ	Результаты работы программы
n = 3 x = 1	$X = (0.937, 0.720, 0.606)$ N = 8 $\omega = 1$ N = 12 $\omega = 1.3$ N = 20 $\omega = 1.6$ N = 98 $\omega = 1.9$
n = 4 x = 2	$X = (8.349, -1.323, -1.851, -2.084)$ N = 22 $\omega = 1$ N = 28 $\omega = 1.3$ N = 51 $\omega = 1.6$ N = 230 $\omega = 1.9$
n = 2 x = 0	$X = (0.000, 0.000)$ N = 1 $\omega = 1$ N = 1 $\omega = 1.3$ N = 1 $\omega = 1.6$ N = 1 $\omega = 1.9$

Выводы

В данной работе необходимо было изучить и реализовать итерационный метод верхней релаксации. Скорость сходимости данного метода зависит от итерационного параметра ω (необходимо $0 < \omega < 2$).

ω	1	1.3	1.6	1.9
СЛАУ 1	628	387	178	366
СЛАУ 2	79	42	34	155
СЛАУ 3	16068	8109	5339	19591
СЛАУ 4	8	12	20	98
СЛАУ 5	22	28	51	230
СЛАУ 6	1	1	1	1

Заметим, что оптимальное значение итерационного параметра ω варьируется в зависимости от СЛАУ. Так в СЛАУ 1-3 оптимальное значение параметра лежит в окрестности 1.6, тогда как в СЛАУ 4-5 оно будет находиться в окрестности 1.

Литература

[1] Костомаров Д.П., Фаворский А.П. Вводные лекции по численным методам: Учеб. Пособие. – М.: Университетская книга, Логос, 2006

[2] Баркалов К.А. Методы параллельных вычислений. Н. Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2011.