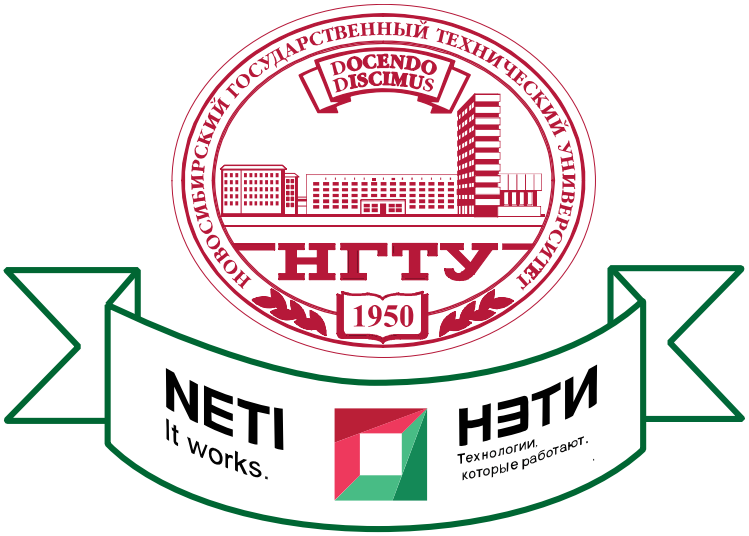
Министерство науки и высшего образования
Российской Федерации

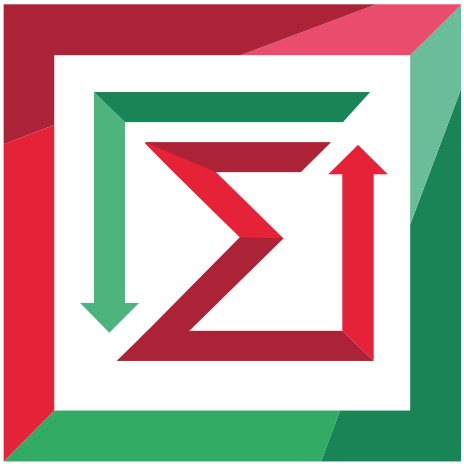
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Новосибирский государственный технический университет»



Кафедра прикладной математики

Практическое задание № 3
по дисциплине «Численные методы»

РЕШЕНИЕ РАЗРЕЖЕННЫХ СЛАУ ТРЕХШАГОВЫМИ



Факультет:	ПМИ
Группа:	ПМ-71
Студент:	Востриков Вячеслав
Вариант:	1
Преподаватели:	Патрушев И.И.
	Задорожный А.Г.
	Персова М.Г.

Новосибирск
2020

1. Цель работы

Изучить особенности реализации трехшаговых итерационных методов для СЛАУ с разреженными матрицами. Исследовать влияние предобуславливания на сходимость изучаемых методов на нескольких матрицах большой (не менее 10000) размерности.

2. Условие задачи

1. Реализовать заданный преподавателем трехшаговый итерационный метод без предобусловливания с учетом следующих требований:
   - матрица A задается в разреженном строчном формате
   - предусмотреть возможность решения СЛАУ большой размерности (не менее 10000). В головной программе резервировать объем памяти, необходимый для хранения исходной матрицы и нужного числа векторов;
   - результат записывать в файл, в процессе счета выдавать на экран сообщение о номере итерации и относительную невязку
2. Протестировать разработанные программы. Для тестирования использовать матрицы небольшой размерности, при этом вектор правой части формировать умножением тестовой матрицы на заданный вектор
3. Сравнить по количеству итераций и времени решения метод Якоби и метод Гаусса- Зейделя с реализованным методом на матрице, построенной по формулам (2.15) (см. п. 3, лаб. раб. № 2) и на матрице с обратным знаком внедиагональных элементов (см. п. 4, лаб. раб. № 2).
4. Повторить п. 3 для плотной матрицы Гильберта для различных размерностей.
5. Повторить п. 3 для матрицы большой размерности, выданной преподавателем.

3. Вариант задания

Метод сопряженных градиентов для симметричной матрицы.

4. Ход работы

- Класс для хранения матрицы в строчном формате

```
class CSR
{
   vector<double> temp; // вектор для работы умножения матрицы на вектор
   public:
   int n;
   int nnz;
   vector<double> values;
   vector<int> row_offsets;
   vector<int> column_indexes;
   CSR(int n, int nnz);
   // конвертация плотной матрицы в разреженную
   void sparsify(const vector<vector<double>> &matrix);
```

```
    vector<double>* mult_by_vector(vector<double> &x);
    // Loading order: values, row_offsets, column_indexes.
    void load(string filename);
};
```

- Класс для представления СЛАУ

```
class SLE
{
    vector<double> z;
    vector<double> r;
    vector<double> r1;
    vector<double> temp;
    public:
    int n;
    int max_iter;
    double tolerance;
    CSR* matrix;
    vector<double> f;
    SLE(int n, int nnz);
    // запуск МСГ
    void steepest_descent(vector<double> &x);
    void load(string matrix_filename,
    string sle_params, string
    f_filename);
};
```

- Подпрограмма для умножения матрицы на вектор

```
vector<double>* CSR::mult_by_vector(vector<double> &x)
{
    for (int i = 0; i < this->n; i++)
    {
        this->temp[i] = 0;
        for (int j = this->row_offsets[i]; j < this->row_offsets[i+1]; j++)
            this->temp[i] += this->values[j] * x[column_indexes[j]];
    }
    return &this->temp;
}
```

- Подпрограмма для метода сопряженных градиентов

```
void SLE::steepest_descent(vector<double>& x)
{
      auto Ax = this->matrix-
      >mult_by_vector(x); subtr(this->f, *Ax,
      this->r);
      copy(this->z, this-
      >r); copy(this->r1,
      this->r);
      double residual = norm(this->r) / norm(this->f);
      for (int iter = 1; iter < max_iter && residual > this->tolerance; iter++)
{
          auto Az = matrix->mult_by_vector(z);
          double alpha = scal_prod(this->r1, this->r1) / scal_prod(*Az, z);
          scale(z, alpha, this->temp);
          add(x, this->temp, x);
          copy(r, r1);
          scale(*Az, alpha, this->temp);
          subtr(this->r1, this->temp, this->r1);
          double beta = scal_prod(this->r1, this->r1) / scal_prod(this->r, this-
>r);
          scale(z, beta, this->temp);
          add(r1, this->temp, z);
          residual = norm(this->r1) / norm(this->f);
          cout << "iter " << iter << " residual " << residual << end
   }
   }
```

Текст программы:

```
void main()
{
    ifstream param("param.txt");
    int n;
    param >> n; int
    nnz; param >>
    nnz;
    auto sle = new SLE(n, nnz);
    sle->load("csr_matrix.txt", "sle_param.txt", "f.txt");
    ifstream x0_file("x0.txt");
    vector<double> x0(n);
    for (int i = 0; i < n; i++)
        x0_file >> x0[i];
    sle->steepest_descent(x0);
    ofstream result("result.txt");
    for (int i = 0; i < n; i++)
       result << x0[i];
}
```

// CSR.h

```
#ifndef CSR_H
#define CSR_H
#include <vector>
#include <string>
#include <fstream>
using namespace std;
class CSR
{
       vector<double> temp;
public:
    int n; int
    nnz;
    vector<double> values;
    vector<int> row_offsets;
    vector<int> column_indexes;
    CSR(int n, int nnz);
    void sparsify(const vector<vector<double>> &matrix);
    vector<double>* mult_by_vector(vector<double> &x);
    // Loading order: values, row_offsets, column_indexes. void
    load(string filename);
};
#endif // !CSR_
```

// CSR.cpp

```
#include "CSR.h"
CSR::CSR(int n, int nnz)
{
    this->n = n; this-
    >nnz = nnz;
    this->values = vector<double>(nnz); this-
    >row_offsets = vector<int>(n + 1);
    this->column_indexes = vector<int>(nnz);
    this->temp = vector<double>(n);
}

vector<double>* CSR::mult_by_vector(vector<double> &x)
{
    for (int i = 0; i < this->n; i++)
    {
       this->temp[i] = 0;
       for (int j = this->row_offsets[i]; j < this->row_offsets[i+1]; j++)
```

```
            this->temp[i] += this->values[j] * x[column_indexes[j]];

    }
    return &this->temp;
}

void CSR::load(string filename)
{
    ifstream file(filename);

    // values loading
    for (int i = 0; i < this->nnz; i++)
    {
       file >> this->values[i];
    }

    // row offsets loading
    for (int i = 0; i < this->n + 1; i++)
    {
       file >> this->row_offsets[i];
    }

    // column indexes loading
    for (int i = 0; i < this->nnz; i++)
    {
       file >> this->column_indexes[i];
    }

    file.close();
}

void CSR::sparsify(const vector<vector<double>>& matrix)
{
    this->column_indexes.clear();
    this->row_offsets.clear();
    this->values.clear();
    this->row_offsets.push_back(0);
    this->nnz = 0;
    for (int i = 0; i < this->n; i++)
    {
        for (int j = 0; j < this->n; j++)
        {
        if (matrix[i][j] != 0)
        {
        this->values.push_back(matrix[i][j]);
        this->column_indexes.push_back(j);

        // Count Number of Non Zero
        // Elements in row i this->nnz++;
        }
        }
        this->row_offsets.push_back(this->nnz);
    }
}
```

// SLE.h

```
#ifndef SLE_H
#define SLE_H
#include "CSR.h"
class SLE
{
    vector<double> z;
```

```
    vector<double> r;
    vector<double> r1;
    vector<double> temp;
    public:
    int n;
    int max_iter; double
    tolerance; CSR*
    matrix;
    vector<double> f;
    SLE(int n, int nnz);
    // ?????? ???
    int steepest_descent(vector<double> &x);
    void load(string matrix_filename,
    string sle_params, string
    f_filename);
};
#endif
```

// SLE.cpp

```
#include "SLE.h"
#include "vector.h"
#include <iostream>

SLE::SLE(int n, int nnz)
{
   this->n = n;
   this->matrix = new CSR (n, nnz);
   this->f = vector<double>(n); this->z
   = vector<double>(n); this->r =
   vector<double>(n); this->r1 =
   vector<double>(n); this->temp =
   vector<double>(n);
}

void SLE::load(string matrix_filename,
string sle_params,
string f_filename)
{
    // matrix loading
    this->matrix->load(matrix_filename);

    // sle params loading
    ifstream sle_file(sle_params);
    sle_file >> this->max_iter;
    sle_file >> this->tolerance;

    // vector f loading
    ifstream f_file(f_filename);
    for (int i = 0; i < this->n; i++)
    {
    f_file >> this->f[i];
    }
}

int SLE::steepest_descent(vector<double>& x)
{
    auto Ax = this->matrix->mult_by_vector(x);
    subtr(this->f, *Ax, this->r);
    copy(this->z, this->r);
    copy(this->r1, this->r);
    double residual = norm(this->r) / norm(this->f);
    int iter;
    for (iter = 1; iter < max_iter && residual > this->tolerance; iter++)
    {
                    auto Az = matrix->mult_by_vector(z);
                    double alpha = scal_prod(this->r1, this->r1) /
                    scal_prod(*Az, z); scale(z, alpha, this->temp);
                    add(x, this-
                    >temp, x);
                    copy(r, r1);
                    scale(*Az, alpha, this->temp);
                    subtr(this->r1, this->temp,
                    this->r1);
                    double beta = scal_prod(this->r1, this->r1) / scal_prod(this->r,
                    this->r);
                    scale(z, beta, this-
       }            >temp); add(r1, this-
                    >temp, z);
                    residual = norm(this->r1) / norm(this->f);
                    cout << "iter " << iter << " residual " << residual << endl;
    return iter;
}
```

Тест программы:

ϵ = 1e-7 (точность получаемого решения)

max_iter = 500

$x_0^T = (0, 0, 0, \ldots, 0)$

№	A	F	X*	X	Кол-во итераций
1	50 6 7 6 50 6 7 6 50	83 124 169	1 2 3	1.0000000000000000 2.0000000000000000 3.0000000000000000	3
2	10 1 2 0 1 20 4 6 2 4 30 3 0 6 3 40	18 77 112 181	1 2 3 4	1.0000000000000000 2.0000000000000000 3.0000000000000000 4.0000000000000009	4
3	20 0 0 15 0 30 0 0 0 35 20 0 15 0 0 20 30 0 35 0 0 20	230 215 60 95 200	1 2 3 4 5	1.0000000000000075 2.0000000000000071 2.9999999999999991 4.0000000000000009 5.0000000000000071	5

5. ***Исследования***:

5.1.Сравним по количеству итераций и времени решения метод Якоби и метод Гаусса- Зейделя с реализованным методом на матрице, построенной по формулам (из л.р.№2) и на матрице с обратным знаком внедиагональных элементов.

а) Матрица с диагональным преобладанием

$$A = \begin{bmatrix} 6 & -3 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ -3 & 6 & -2 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 10 & -4 & 0 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 0 & -4 & 7 & -1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & -1 & 5 & -2 & 0 & 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & -2 & 8 & -3 & 0 & 0 & 0 & 0 & -3 \\ -2 & 0 & 0 & 0 & 0 & -3 & 6 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & -1 & 5 & -3 & 0 & 0 & 0 \\ 0 & 0 & -4 & 0 & 0 & 0 & 0 & -3 & 8 & -1 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & -4 & 7 & -4 & 0 \\ 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & -1 & 7 & -1 \\ 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 & -1 & 4 \end{bmatrix}$$

Для методов Якоби и Гаусса – Зейделя параметр релаксации берем за 1.8

№	Метод	Затраченное время, сек.	Количество итераций
1	Якоби	0.05	103
2	Гаусс-Зейдель	0.07	103
3	МСГ	0.013	12

5.2. Матрица с диагональным преобладанием и обратным знаком внедиагональных элементом

$$B = \begin{bmatrix} 6 & 3 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 6 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 10 & 4 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 4 & 7 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 5 & 2 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 & 8 & 3 & 0 & 0 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 & 3 & 6 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 5 & 3 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 3 & 8 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 4 & 7 & 4 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 7 & 1 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 1 & 4 \end{bmatrix}$$

Для методов Якоби и Гаусса – Зейделя параметр релаксации берем за 1.5

№	Метод	Затраченное время, сек.	Количество итераций
1	Якоби	0.04	43
2	Гаусс-Зейдель	0.04	43
3	МСГ	0.028	12

5.3.Проведем тестирование программы на матрицах Гильберта различной размерности

$$H_{ij} = \frac{1}{i+j-1}, i, j = 1, 2, 3, \ldots, n$$

ϵ = 1e-7 (точность получаемого решения)
max_iter = 500

$x^* = (1,1,1, \ldots ,1)$

$x^T = (0, 0, 0, \ldots, 0)$

№	N	Количество итераций
1	2	2
2	5	5
3	10	7
4	15	7
5	50	10
6	100	12
7	200	11
8	300	13
9	600	16

5.4.Проведем тестирование программы на матрицах большой размерности (тесты из DiSpace) для метода сопряженных градиентов

№	n	Затраченное время, сек.	Количество итераций
1	945	0.688	392
2	4545	9.239	2007

6. Вывод:

В результате сравнения МСГ, метода Якоби и метода Гаусса-Зейделя было показано, что МСГ выполняется за гораздо меньшее число итераций и меньшее время.