|  |  |  |
| --- | --- | --- |
| 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****ИТЕРАЦИОННЫМИ МЕТОДАМИ** | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМ-71 |
| Студент: | Востриков Вячеслав |
| Вариант: | 1 |
| Преподаватели: | Патрушев И.И. |
|  | Задорожный А.Г. |
|  | Персова М.Г. |
|  |  |
|  | | |
| Новосибирск | | |
| 2020 | | |

# 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.

1. **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.
2. **Вариант задания**

Метод сопряженных градиентов для симметричной матрицы.

1. **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++)

{

>r);

}

}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-

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

𝑥𝑇 = (0, 0, 0, . . . , 0)

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 30  0 20 0 0 35  0 0 20 0 0  15 0 0 20 0  30 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 |

1. ***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

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

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

ϵ = 1e-7 (точность получаемого решения) max\_iter = 500

𝑥∗ = (1,1,1, … ,1)

𝑥𝑇 =(0, 0, 0, . . . , 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 |
|  |  |  |  |

1. **Вывод:**

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