|  |  |  |
| --- | --- | --- |
|  | | |
| 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 С ПРЕДОБУСЛОВЛИВАНИЕМ | | |
|  | | |
|  |  |  |
| Группа ПМ-12 | Лойченко данила, Овчинников иван |
| Вариант 9 |  |
|  |  |
|  |  |
|  |  |
| Преподаватели | задорожный александр генадьевич |
|  |  |
| Новосибирск, 2023 | | |

1. **Задание**

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

1. **Математическая модель**

Алгоритм метода сопряженных градиентов для системы уравнений с несимметричной матрицей А:



Заключается в симметризации СЛАУ следующим образом:



Тогда начальное приближение выбирается следующим образом:





Далее для  производятся следующие вычисления:











где  - вектор начальное приближения; - вектор решения на -й (текущей итерации); - вектор невязки на -й (текущей) итерации; - вектор спуска (сопряженное направление) на -й итерации;  - коэффициенты.

Выход из итерационного процесса осуществляется либо по условию малости относительной невязки



Для ускорения сходимости итерационных методов обычно используют предобусловливание матрицы системы. Одним методов предобусловливания является метод **неполной факторизации.** Рассмотрим процедуру для предобусловленной СЛАУ , построенную на основе метода сопряженных градиентов. Итак вместо исходной СЛАУ  будем решать СЛАУ , в которой



Где матрицы  и  соответственно нижняя треугольная и верхнетругольная матрицы неполной факторизации исходной матрицы .

Тогда формулы метода сопряженных градиентов преобразуются к следующему виду:

Выбирается начальное приближение и полагается













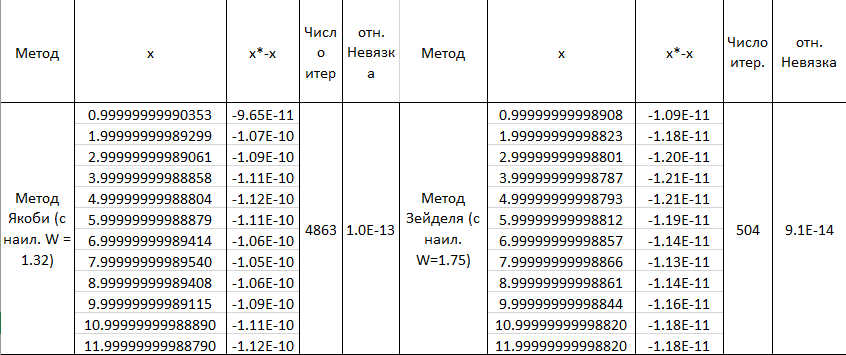


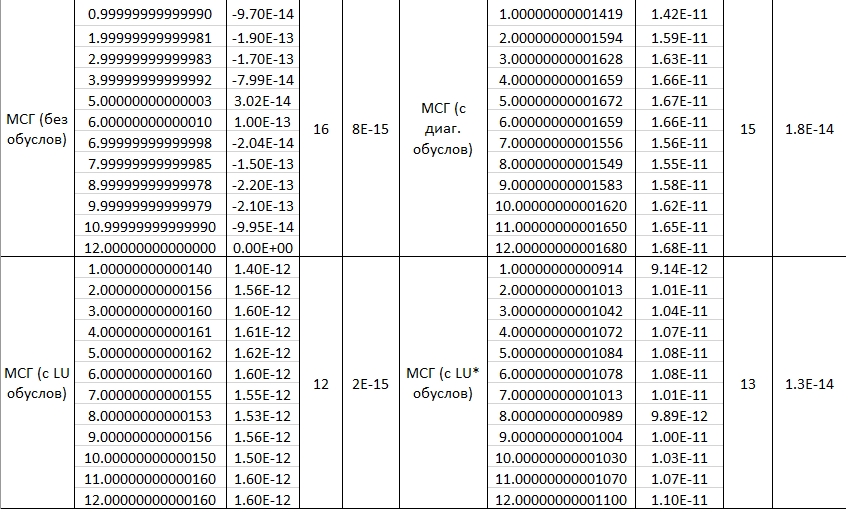
По окончании итерационного процесса вектор решения вычисляется следующим образом:

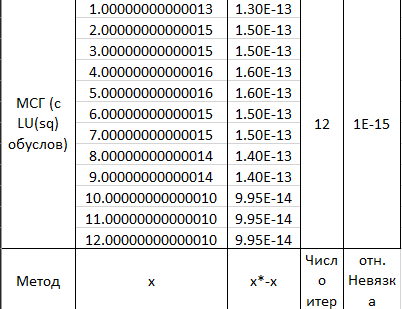


1. **Исследование матриц с регулируемым числом обусловленности**

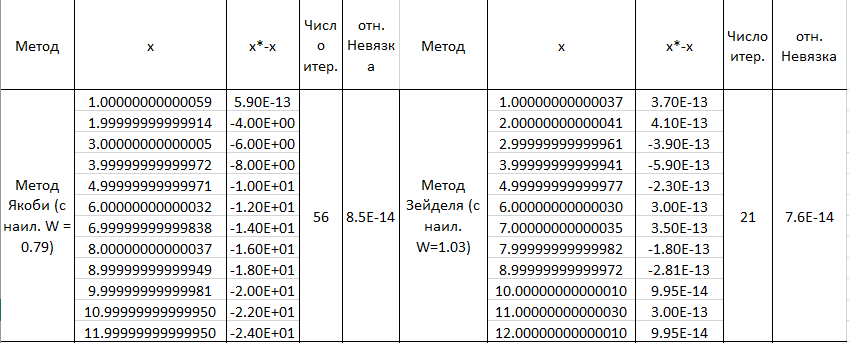
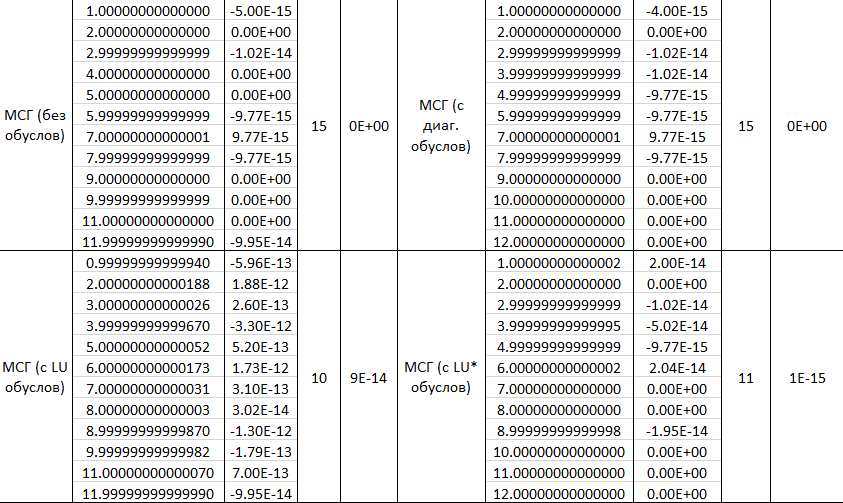
****

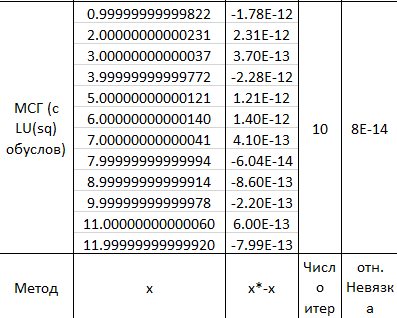








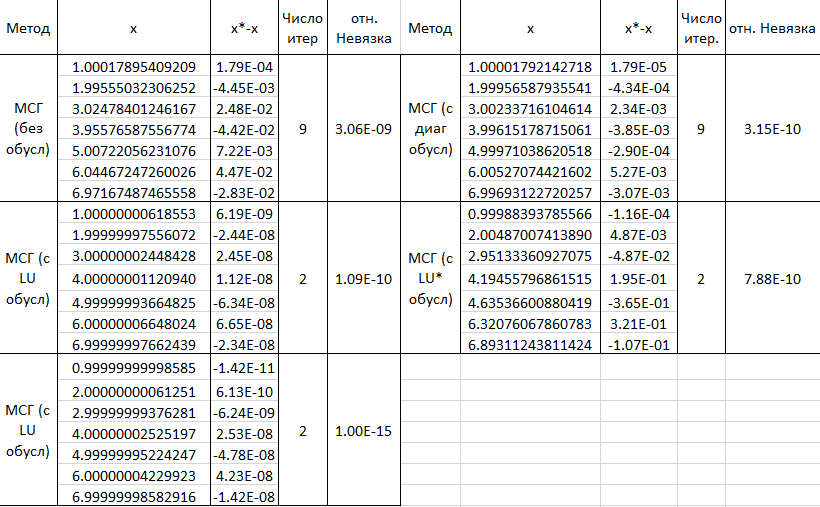
 

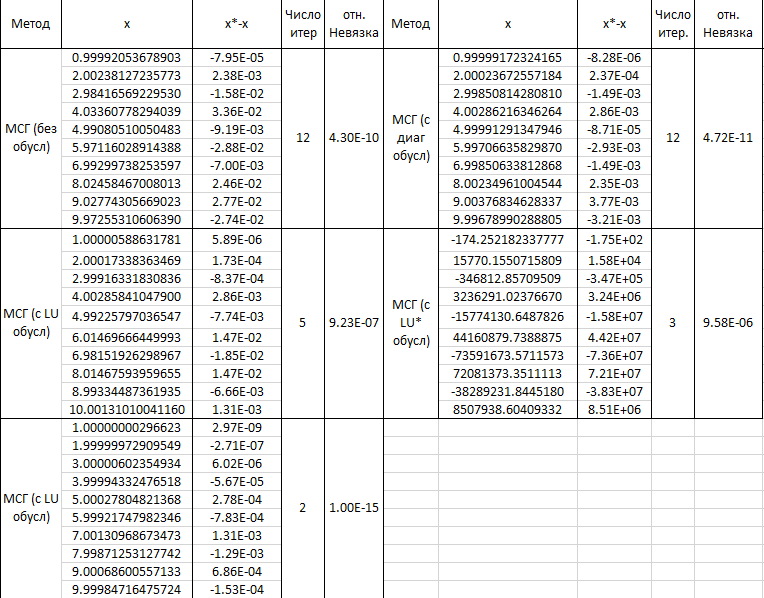


**Исследование матриц большой размерности**

|  |  |
| --- | --- |
| N=4545 | N=945 |
|  |  |

**Исследование матриц Гильберта**





**Подсчет количества действий**

Несимметричного МСГ без предобусловливания

****

Итого количество действий составляет:

МСГ без предобусловливания: 

Несимметричного МСГ c диагональным предобусловливанием



Обозначим вектор обусловливания как 

Тогда



Итого количество действий составляет:

МСГ c диагональным предобусловливанием: 

Вычислим для начала количество действий для LU(sq) разложения и прямого и обратного хода для него

LU(sq) разложение:



Прямой или обратный ход



Несимметричного МСГ с LU(sq) предобусловливанием



Итого количество действий составляет:

МСГ с LU(sq) предобусловливанием: 

**Выводы**

1. Исследование на матрице с отрицательными внедиагональными элементами привело к следующему выводу - все виды МСГ справились с решением СЛАУ в разы за меньшее число итераций и в большинстве случаев точнее, чем методы Якоби и Зейделя.
2. Исследование на матрице с положительными внедиагональными элементами привело к следующему выводу - все итерационные методы показали хороший результат. Количество итераций и относительная невязка у всех (за исключением Якоби в нем кол-во итераций было в 2-3 раза больше, чем у других) методов находятся на достаточно близком уровне.
3. Исследование на матрице большой размерности привело следующему выводу - факторизация в разы ускоряет решение СЛАУ. Хотя и для этого требуются высчитать LU разложение, у которого сложность , но если необходимо посчитать множество СЛАУ с одинаковой матрицей А, но разной правой частью, то затраты на подсчитывание LU разложения будут оправданы.
4. Исследование на матрицах Гильберта привело к следующему выводу - использование предобусловливания LU-разложением на матрицах с большим числом обусловленности – является плохой идеей. Т.к. матрица Гильберта полная, то МСГ с LU предобусловливанием должен решать СЛАУ на первой итерации, но поскольку матрица плохо обусловлена, то LU вычисляется с большой погрешностью и поэтому итерационный процесс продолжается на ещё несколько итераций, чтобы добиться требуемой точности.
5. Подсчет количества действий привёл к следующим выводам

Если дана матрица с отрицательными, внедиагональных элементов, то метод МСГ будет на порядок менее вычислительно затратным, чем Зейдель



Если дана матрица положительными элементами, то метод Зейдель будет на порядок менее вычислительно затратным, чем МСГ



Так что выбор итерационного процесса во многом основывается на том для какой матрицы будет искаться вектор неизвестных

**Приложение**

**SLAE.h**

#pragma once

#pragma once

#include <cstdio>

#include <math.h>

#define REALOUTD "%.15f\t"

class SLAE {

public:

int n, maxiter = 10000, nProfile = 0;

double eps = 1e-13;

double\* al, \* au, \* di;

double\* alLU, \* auLU, \* diLU;

double\* x, \* x0, \* b, \* xtrue, \* dP;

double\* r, \* z, \* tmp1;

int\* ia, \* ja;

void Input(FILE\* paramf, FILE\* iaf, FILE\* jaf, FILE\* alf, FILE\* auf, FILE\* dif, FILE\* bf);

void MatrixVectorMultiplication(double\* vectorMult, double\* vectorOut);

void TransposedMatrixVectorMultiplication(double\* vectorMult, double\* vectorOut);

double CalculateRelativeDiscrepancyWithR(double norm);

double CalculateRelativeDiscrepancy(double norm);

void MethodOfConjugateGradientsForSymMatrix();

void MethodOfConjugateGradientsForNonSymMatrix();

void MethodOfConjugateGradientsForSymMatrixWithDiagP();

void MethodOfConjugateGradientsForNonSymMatrixWithDiagP();

void MethodOfConjugateGradientsForSymMatrixWithLuP();

void MethodOfConjugateGradientsForNonSymMatrixWithLuP();

void MethodOfConjugateGradientsForNonSymMatrixWithLuAsterP();

void MethodOfConjugateGradientsForNonSymMatrixWithLuSqP();

void VectorConditionalityForSymMatrixDiagP(double\* vectorIn, double\* vectorOut);

void VectorConditionalityForNonSymMatrixDiagP(double\* vectorIn, double\* vectorOut);

void CalculateLU();

void CalculateLUaster();

void CalculateLUsq();

void GenerateHilbertMatrix(int size);

void SolveForward(double\* lowerTringMat, double\* diag, double\* rightVector, double\* vectorX);

void SolveBackward(double\* upperTringMat, double\* diag, double\* rightVector, double\* vectorX);

void SolveForwardLU(double\* lowerTringMat, double\* rightVector, double\* vectorX);

void SolveBackwardLU(double\* upperTringMat, double\* rightVector, double\* vectorX);

void SolveForwardLU(double\* lowerTringMat, double\* diag, double\* rightVector, double\* vectorX);

void SolveBackwardLU(double\* upperTringMat, double\* diag, double\* rightVector, double\* vectorX);

void MatrixUVectorMultiplicationLU(double\* U, double\* vectorMult, double\* vectorOut);

void MatrixUVectorMultiplicationLU(double\* U, double \*diag, double\* vectorMult, double\* vectorOut);

void VectorUVectorMultiplication(double\* x);

void CalculateZ\_LU(double\* vectorOut);

void CalculateZ\_LUaster(double\* vectorOut);

void CalculateZ\_LUsq(double\* vectorOut);

void CalculateXkRk(double ak);

void CalculateZk(double bk);

void Calculate\_dP();

void CalculateRelativeDiscrepancy(double\* vectorMult, double\* vectorOut);

void VectorSubtract(double\* first, double\* second, double\* result);

double VectorScalarProduction(double\* vector);

double VectorScalarProduction(double\* vector1, double\* vector2);

double VectorNorm(double\* vector);

void VectorCopy(double\* first, double\* second);

void OutputDense();

void OutputLUDense();

void VectorOutputSolution(FILE\* out);

protected:

void AllocateMemory();

void ClearMemory();

};

SLAE.cpp

#include "slae.h"

void SLAE::Input(FILE\* paramf, FILE\* iaf, FILE\* jaf, FILE\* alf, FILE\* auf, FILE\* dif, FILE\* bf) {

fscanf\_s(paramf, "%d", &n);

fscanf\_s(paramf, "%d", &maxiter);

fscanf\_s(paramf, "%lf", &eps);

ia = new int[n + 1];

for (int i = 0; i <= n; i++)

fscanf\_s(iaf, "%d", &ia[i]);

nProfile = ia[n] - ia[0];

AllocateMemory();

for (int i = 0; i < nProfile; i++)

fscanf\_s(jaf, "%d", &ja[i]);

if (ia[0]) {

for (int i = 0; i <= n; i++)

ia[i]--;

for (int i = 0; i < nProfile; i++)

ja[i]--;

}

for (int i = 0; i < nProfile; i++)

fscanf\_s(alf, "%lf", &al[i]);

for (int i = 0; i < nProfile; i++)

fscanf\_s(auf, "%lf", &au[i]);

for (int i = 0; i < n; i++)

fscanf\_s(dif, "%lf", &di[i]);

for (int i = 0; i < n; i++)

fscanf\_s(bf, "%lf", &b[i]);

//for (int i = 0; i < n; i++)

// x0[i] = 0;

for (int i = 0; i < n; i++)

{

xtrue[i] = i + 1;

}

}

void SLAE::OutputDense()

{

int flagfound = 0;

for (int i = 0; i < n; i++)

{

int k = ia[i + 1] - ia[i];

if (k == 0)

{

for (int j = 0; j < i; j++)

{

printf(REALOUTD, 0.0);

}

}

else

{

int lastj = 0;

for (int j = ia[i]; j < ia[i + 1]; j++) //��� ��� 100 ���� ���������.

{

for (int p = lastj; p < ja[j]; p++) //��� ��� 100 ���� ���������.

{

printf(REALOUTD, 0.0);

}

printf(REALOUTD, al[j]);

lastj = ja[j] + 1;

}

for (int j = lastj; j < i; j++) //??

{

printf(REALOUTD, 0.0);

}

}

printf(REALOUTD, di[i]);

for (int j = i + 1; j < n; j++)

{

k = ia[j + 1] - ia[j];

if (k == 0) {

printf(REALOUTD, 0.0);

}

else

{

flagfound = 0;

for (k = ia[j]; k < ia[j + 1]; k++)

{

if (ja[k] == i)

{

printf(REALOUTD, au[k]);

flagfound = 1;

break;

}

}

if (flagfound == 0)

printf(REALOUTD, 0.0);

}

}

printf("\n");

}

printf("\n");

}

void SLAE::OutputLUDense()

{

int flagfound = 0;

for (int i = 0; i < n; i++)

{

int k = ia[i + 1] - ia[i];

if (k == 0)

{

for (int j = 0; j < i; j++)

{

printf(REALOUTD, 0.0);

}

}

else

{

int lastj = 0;

for (int j = ia[i]; j < ia[i + 1]; j++) //��� ��� 100 ���� ���������.

{

for (int p = lastj; p < ja[j]; p++) //��� ��� 100 ���� ���������.

{

printf(REALOUTD, 0.0);

}

printf(REALOUTD, alLU[j]);

lastj = ja[j] + 1;

}

for (int j = lastj; j < i; j++) //??

{

printf(REALOUTD, 0.0);

}

}

printf(REALOUTD, diLU[i]);

for (int j = i + 1; j < n; j++)

{

k = ia[j + 1] - ia[j];

if (k == 0) {

printf(REALOUTD, 0.0);

}

else

{

flagfound = 0;

for (k = ia[j]; k < ia[j + 1]; k++)

{

if (ja[k] == i)

{

printf(REALOUTD, auLU[k]);

flagfound = 1;

break;

}

}

if (flagfound == 0)

printf(REALOUTD, 0.0);

}

}

printf("\n");

}

printf("\n");

}

void SLAE::MatrixVectorMultiplication(double\* vectorMult, double\* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = di[i] \* vectorMult[i];

for (int k = ia[i]; k < ia[i + 1]; k++)

{

int j = ja[k];

vectorOut[i] += al[k] \* vectorMult[j];

vectorOut[j] += au[k] \* vectorMult[i];

}

}

}

void SLAE::TransposedMatrixVectorMultiplication(double\* vectorMult, double\* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = di[i] \* vectorMult[i];

for (int k = ia[i]; k < ia[i + 1]; k++)

{

int j = ja[k];

vectorOut[i] += au[k] \* vectorMult[j];

vectorOut[j] += al[k] \* vectorMult[i];

}

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ������������ ������� \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void SLAE::MethodOfConjugateGradientsForSymMatrix()

{

VectorCopy(x0, x);

CalculateRelativeDiscrepancy(x0, tmp1);

VectorCopy(tmp1, r);

VectorCopy(r, z);

double normB = VectorNorm(b);

double ak = 0, bk = 0;

double r\_rPrev = VectorScalarProduction(r); // ( r(k-1) , r(k-1) )

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

MatrixVectorMultiplication(z, tmp1); // A\*z(k-1)

double Az\_zPrev = VectorScalarProduction(tmp1, z); //( A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Az\_zPrev;

CalculateXkRk(ak);

double r\_rCur = VectorScalarProduction(r); // ( rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

}

void SLAE::MethodOfConjugateGradientsForSymMatrixWithDiagP()

{

VectorCopy(x0, x);

CalculateRelativeDiscrepancy(x0, tmp1);

VectorCopy(tmp1, r);

VectorConditionalityForSymMatrixDiagP(tmp1, z);

VectorConditionalityForSymMatrixDiagP(r, tmp1);

double r\_rPrev = VectorScalarProduction(tmp1, r); //( M^(-1)\*r(k-1) , r(k-1) )

double normB = VectorNorm(b);

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

double ak = 0, bk = 0;

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

//VectorConditionalityForSymMatrixDiagP(r, tmp1);

//double r\_rPrev = VectorScalarProduction(tmp1, r); //( M^(-1)\*r(k-1) , r(k-1) )

MatrixVectorMultiplication(z, tmp1); //A\*z(k-1)

double Az\_zPrev = VectorScalarProduction(tmp1, z); // ( A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Az\_zPrev;

CalculateXkRk(ak);

VectorConditionalityForSymMatrixDiagP(r, tmp1); // M^(-1)\*rk

double r\_rCur = VectorScalarProduction(tmp1, r); //( M^(-1)\*rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

}

void SLAE::MethodOfConjugateGradientsForSymMatrixWithLuP()

{

VectorCopy(x0, x);

CalculateRelativeDiscrepancy(x0, r);

// M = LU

// M^(-1) = U^(-1)\*L^(-1)

SolveForwardLU(alLU, diLU, r, tmp1);

SolveBackwardLU(auLU, tmp1, z);

double normB = VectorNorm(b);

double ak = 0, bk = 0;

//ak

SolveForwardLU(alLU, diLU, r, tmp1);

SolveBackwardLU(auLU, tmp1, tmp1);

double Mr\_rPrev = VectorScalarProduction(tmp1, r); // ( M^(-1)\*r(k-1) , r(k-1) )

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

MatrixVectorMultiplication(z, tmp1); // A\*z(k-1)

double Az\_zPrev = VectorScalarProduction(tmp1, z); //( A\*z(k-1) , z(k-1) )

ak = Mr\_rPrev / Az\_zPrev;

CalculateXkRk(ak);

SolveForwardLU(alLU, diLU, r, tmp1);

SolveBackwardLU(auLU, tmp1, tmp1);

double Mr\_rCur = VectorScalarProduction(tmp1, r); // ( rk , rk )

bk = Mr\_rCur / Mr\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

Mr\_rPrev = Mr\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void SLAE::MethodOfConjugateGradientsForNonSymMatrix()

{

VectorCopy(x0, x);

CalculateRelativeDiscrepancy(x0, tmp1);

TransposedMatrixVectorMultiplication(tmp1, r);

VectorCopy(r, z);

double normB = VectorNorm(b);

double ak = 0, bk = 0;

double r\_rPrev = VectorScalarProduction(r); // ( r(k-1) , r(k-1) )

double r\_rCur, Az\_zPrev;

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

MatrixVectorMultiplication(z, x0); // A\*z(k-1)

TransposedMatrixVectorMultiplication(x0, tmp1);

Az\_zPrev = VectorScalarProduction(tmp1, z); //( A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Az\_zPrev;

CalculateXkRk(ak);

r\_rCur = VectorScalarProduction(r); // ( rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

printf("%.15lf\n", CalculateRelativeDiscrepancy(normB));

}

void SLAE::MethodOfConjugateGradientsForNonSymMatrixWithDiagP()

{

Calculate\_dP();

VectorConditionalityForNonSymMatrixDiagP(x0, x);

VectorCopy(x0, x);

VectorUVectorMultiplication(x);

CalculateRelativeDiscrepancy(x0, tmp1);

VectorConditionalityForNonSymMatrixDiagP(tmp1, tmp1);

VectorConditionalityForNonSymMatrixDiagP(tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

VectorConditionalityForNonSymMatrixDiagP(x0, x0);

VectorCopy(x0, r);

VectorCopy(x0, z);

double normB = VectorNorm(b);

double ak = 0, bk = 0;

double r\_rPrev = VectorScalarProduction(r); // ( r(k-1) , r(k-1) )

double r\_rCur, Az\_zPrev;

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

VectorConditionalityForNonSymMatrixDiagP(z, tmp1);

MatrixVectorMultiplication(tmp1, x0); // A\*z(k-1)

VectorConditionalityForNonSymMatrixDiagP(x0, x0);

VectorConditionalityForNonSymMatrixDiagP(x0, x0);

TransposedMatrixVectorMultiplication(x0, tmp1); // A^T\*A\*z(k - 1)

VectorConditionalityForNonSymMatrixDiagP(tmp1, tmp1);

Az\_zPrev = VectorScalarProduction(tmp1, z); // ( A^T\*A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Az\_zPrev;

CalculateXkRk(ak);

r\_rCur = VectorScalarProduction(r); // ( rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

VectorConditionalityForNonSymMatrixDiagP(x, x);

printf("%.15lf\n", CalculateRelativeDiscrepancy(normB));

}

void SLAE::MethodOfConjugateGradientsForNonSymMatrixWithLuP()

{

MatrixUVectorMultiplicationLU(auLU, x0, x);

CalculateRelativeDiscrepancy(x0, tmp1);

SolveForwardLU(alLU, diLU, tmp1, tmp1);

SolveBackwardLU(alLU, diLU, tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(auLU, x0, tmp1);

VectorCopy(tmp1, r);

VectorCopy(tmp1, z);

double r\_rPrev, r\_rCur, Newz\_zPrev, ak, bk;

r\_rPrev = VectorScalarProduction(r); // ( r(k-1) , r(k-1) )

double normB = VectorNorm(b);

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

VectorCopy(z, tmp1);

CalculateZ\_LU(tmp1); // U^(-T)\*A^T\*L^(-T)\*L^(-1)\*A\*U^(-1)\*z(k-1)

Newz\_zPrev = VectorScalarProduction(tmp1, z); //( A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Newz\_zPrev;

CalculateXkRk(ak);

r\_rCur = VectorScalarProduction(r); // ( M^(-1)\*rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

SolveBackwardLU(auLU, x, x);

printf("%.15lf\n", CalculateRelativeDiscrepancy(normB));

}

void SLAE::MethodOfConjugateGradientsForNonSymMatrixWithLuAsterP()

{

MatrixUVectorMultiplicationLU(auLU, diLU, x0, x);

CalculateRelativeDiscrepancy(x0, tmp1);

SolveForwardLU(alLU, tmp1, tmp1);

SolveBackwardLU(alLU, tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(auLU, diLU, x0, tmp1);

VectorCopy(tmp1, r);

VectorCopy(tmp1, z);

double r\_rPrev, r\_rCur, Newz\_zPrev, ak, bk;

r\_rPrev = VectorScalarProduction(r); // ( r(k-1) , r(k-1) )

double normB = VectorNorm(b);

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

VectorCopy(z, tmp1);

CalculateZ\_LUaster(tmp1); // U^(-T)\*A^T\*L^(-T)\*L^(-1)\*A\*U^(-1)\*z(k-1)

Newz\_zPrev = VectorScalarProduction(tmp1, z); //( A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Newz\_zPrev;

CalculateXkRk(ak);

r\_rCur = VectorScalarProduction(r); // ( M^(-1)\*rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

SolveBackwardLU(auLU, diLU, x, x);

printf("%.15lf\n", CalculateRelativeDiscrepancy(normB));

}

void SLAE::MethodOfConjugateGradientsForNonSymMatrixWithLuSqP()

{

MatrixUVectorMultiplicationLU(auLU, x0, x);

CalculateRelativeDiscrepancy(x0, tmp1);

SolveForwardLU(alLU, diLU, tmp1, tmp1);

SolveBackwardLU(alLU, diLU, tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(auLU, diLU, x0, tmp1);

VectorCopy(tmp1, r);

VectorCopy(tmp1, z);

double r\_rPrev, r\_rCur, Newz\_zPrev, ak, bk;

r\_rPrev = VectorScalarProduction(r); // ( r(k-1) , r(k-1) )

double normB = VectorNorm(b);

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

VectorCopy(z, tmp1);

CalculateZ\_LUsq(tmp1); // U^(-T)\*A^T\*L^(-T)\*L^(-1)\*A\*U^(-1)\*z(k-1)

Newz\_zPrev = VectorScalarProduction(tmp1, z); //( A\*z(k-1) , z(k-1) )

ak = r\_rPrev / Newz\_zPrev;

CalculateXkRk(ak);

r\_rCur = VectorScalarProduction(r); // ( M^(-1)\*rk , rk )

bk = r\_rCur / r\_rPrev;

CalculateZk(bk);

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

SolveBackwardLU(auLU, diLU, x, x);

printf("%.15lf\n", CalculateRelativeDiscrepancy(normB));

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void SLAE::VectorUVectorMultiplication(double \*x)

{

for (int i = 0; i < n; i++)

{

x[i] = x[i] \* dP[i];

}

}

void SLAE::CalculateXkRk(double ak)

{

for (int i = 0; i < n; i++)

{

x[i] = x[i] + ak \* z[i]; // xk = x(k-1) + ak\*z(k-1)

r[i] = r[i] - ak \* tmp1[i]; // rk = r(k-1) - ak\*U^(-T)\*A^T\*L^(-T)\*L^(-1)\*A\*U^(-1)\*z(k-1)

}

}

void SLAE::CalculateZk(double bk)

{

for (int i = 0; i < n; i++) // zk = M^(-1)\*rk + bk\*z(k-1)

z[i] = r[i] + bk \* z[i];

}

void SLAE::CalculateLU()

{

for (int i = 0; i < n; i++) //i = 7

{

int i0 = ia[i]; //15

int i1 = ia[i + 1]; //19

double sumD = 0;

for (int k = i0; k < i1; k++)

{

double sumL = 0, sumU = 0;

int j = ja[k];

int j0 = ia[j];

int j1 = ia[j + 1];

int ik = i0;

int kj = j0;

for (; ik < k and kj < j1; )

{

if (ja[ik] < ja[kj])

ik++;

if (ja[ik] > ja[kj])

kj++;

if (ja[ik] == ja[kj]) {

sumL += alLU[ik] \* auLU[kj];

sumU += alLU[kj] \* auLU[ik];

ik++; kj++;

}

}

alLU[k] = al[k] - sumL;

auLU[k] = (au[k] - sumU) / diLU[j];

sumD += alLU[k] \* auLU[k];

}

diLU[i] = di[i] - sumD;

}

}

void SLAE::CalculateLUaster()

{

for (int i = 0; i < n; i++) //i = 7

{

int i0 = ia[i]; //15

int i1 = ia[i + 1]; //19

double sumD = 0;

for (int k = i0; k < i1; k++)

{

double sumL = 0, sumU = 0;

int j = ja[k];

int j0 = ia[j];

int j1 = ia[j + 1];

int ik = i0;

int kj = j0;

for (; ik < k and kj < j1; )

{

if (ja[ik] < ja[kj])

ik++;

if (ja[ik] > ja[kj])

kj++;

if (ja[ik] == ja[kj]) {

sumL += alLU[ik] \* auLU[kj];

sumU += alLU[kj] \* auLU[ik];

ik++; kj++;

}

}

alLU[k] = (al[k] - sumL) / diLU[j];

auLU[k] = (au[k] - sumU);

sumD += alLU[k] \* auLU[k];

}

diLU[i] = di[i] - sumD;

}

}

void SLAE::CalculateLUsq()

{

for (int i = 0; i < n; i++) //i = 7

{

int i0 = ia[i]; //15

int i1 = ia[i + 1]; //19

double sumD = 0;

for (int k = i0; k < i1; k++)

{

double sumL = 0, sumU = 0;

int j = ja[k];

int j0 = ia[j];

int j1 = ia[j + 1];

int ik = i0;

int kj = j0;

for (; ik < k and kj < j1; )

{

if (ja[ik] < ja[kj])

ik++;

if (ja[ik] > ja[kj])

kj++;

if (ja[ik] == ja[kj]) {

sumL += alLU[ik] \* auLU[kj];

sumU += alLU[kj] \* auLU[ik];

ik++; kj++;

}

}

alLU[k] = (al[k] - sumL) / diLU[j];

auLU[k] = (au[k] - sumU) / diLU[j];

sumD += alLU[k] \* auLU[k];

}

diLU[i] = sqrt(di[i] - sumD);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void SLAE::GenerateHilbertMatrix(int size)

{

n = size;

for (int i = 0; i < size; i++)

{

nProfile += i;

}

ia = new int[n + 1];

AllocateMemory();

ia[0] = 0;

for (int i = 1, k = 0; i <= n; i++)

{

ia[i] = ia[i - 1] + (i - 1);

di[i - 1] = (double)1 / (2 \* i - 1);

for (int j = 1; j < i; j++, k++)

{

al[k] = (double)1 / (i + j - 1);

au[k] = (double)1 / (i + j - 1);

ja[k] = j - 1;

}

}

for (int i = 0; i < n; i++)

{

double sum = 0;

for (int xk = 1; xk <= n; xk++)

{

sum += (double)1 / (i + xk) \* xk;

}

b[i] = sum;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void SLAE::CalculateRelativeDiscrepancy(double \* vectorMult, double \* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = di[i] \* vectorMult[i];

vectorOut[i] = b[i] - vectorOut[i];

for (int k = ia[i]; k < ia[i + 1]; k++)

{

int j = ja[k];

vectorOut[i] += al[k] \* vectorMult[j];

vectorOut[j] += au[k] \* vectorMult[i];

}

}

}

void SLAE::VectorConditionalityForSymMatrixDiagP(double\* vectorIn, double\* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = vectorIn[i] / di[i];

}

}

void SLAE::Calculate\_dP()

{

for (int i = 0; i < n; i++)

{

dP[i] = sqrt(di[i]);

}

}

void SLAE::VectorConditionalityForNonSymMatrixDiagP(double\* vectorIn, double\* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = vectorIn[i] / dP[i];

}

}

void SLAE::VectorSubtract(double\* first, double\* second, double\* result)

{

for (int i = 0; i < n; i++)

{

result[i] = first[i] - second[i];

}

}

void SLAE::VectorCopy(double\* from, double\* to)

{

for (int i = 0; i < n; i++)

{

to[i] = from[i];

}

}

double SLAE::VectorScalarProduction(double\* vector1, double\* vector2)

{

double prod = 0;

for (int i = 0; i < n; i++)

{

prod += vector1[i] \* vector2[i];

}

return prod;

}

double SLAE::VectorScalarProduction(double\* vector)

{

double prod = 0;

for (int i = 0; i < n; i++)

{

prod += vector[i] \* vector[i];

}

return prod;

}

double SLAE::VectorNorm(double\* vector)

{

double norm = 0;

for (int i = 0; i < n; i++)

{

norm += vector[i] \* vector[i];

}

return sqrt(norm);

}

void SLAE::VectorOutputSolution(FILE\* out)

{

for (int i = 0; i < n; i++)

{

fprintf(out, "%.15lf\n", x[i]);

}

printf("\n");

ClearMemory();

}

void SLAE::SolveForwardLU(double\* lowerTringMat,double \*diag, double\* rightVector, double\* vectorX) {

for (int i0, i1, i = 0; i < n; i++)

{

double sum = 0;

i0 = ia[i];

i1 = ia[i + 1];

//int j = i - (i1 - i0);

for (int k = i0; k < i1; k++)

{

int j = ja[k];

sum += lowerTringMat[k] \* vectorX[j];

}

vectorX[i] = (rightVector[i] - sum) / diag[i];

}

}

void SLAE::SolveForwardLU(double\* lowerTringMat, double\* rightVector, double\* vectorX) {

for (int i0, i1, i = 0; i < n; i++)

{

double sum = 0;

i0 = ia[i];

i1 = ia[i + 1];

//int j = i - (i1 - i0);

for (int k = i0; k < i1; k++)

{

int j = ja[k];

sum += lowerTringMat[k] \* vectorX[j];

}

vectorX[i] = (rightVector[i] - sum);

}

}

void SLAE::SolveBackwardLU(double\* upperTringMat, double\* diag, double\* rightVector, double\* vectorX) {

for (int i0, i1, i = n - 1; i >= 0; i--)

{

i0 = ia[i];

i1 = ia[i + 1];

vectorX[i] = rightVector[i] / diag[i];

for (int j, k = i0; k < i1; k++)

{

j = ja[k];

rightVector[j] -= upperTringMat[k] \* vectorX[i];

}

}

}

void SLAE::SolveBackwardLU(double\* upperTringMat, double\* rightVector, double\* vectorX) {

for (int i0, i1, i = n - 1; i >= 0; i--)

{

i0 = ia[i];

i1 = ia[i + 1];

vectorX[i] = rightVector[i];

for (int j, k = i0; k < i1; k++)

{

j = ja[k];

rightVector[j] -= upperTringMat[k] \* vectorX[i];

}

}

}

void SLAE::MatrixUVectorMultiplicationLU(double\* upperTringMat, double\* vectorMult, double\* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = vectorMult[i];

for (int j, k = ia[i]; k < ia[i + 1]; k++)

{

j = ja[k];

vectorOut[j] += upperTringMat[k] \* vectorMult[i];

}

}

}

void SLAE::MatrixUVectorMultiplicationLU(double\* upperTringMat, double\* diag, double\* vectorMult, double\* vectorOut)

{

for (int i = 0; i < n; i++)

{

vectorOut[i] = vectorMult[i]\*diag[i];

for (int j, k = ia[i]; k < ia[i + 1]; k++)

{

j = ja[k];

vectorOut[j] += upperTringMat[k] \* vectorMult[i];

}

}

}

void SLAE::CalculateZ\_LU(double\* vectorOut)

{

SolveBackwardLU(auLU, tmp1, tmp1);

MatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(alLU, diLU, x0, tmp1);

SolveBackwardLU(alLU, diLU, tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(auLU, x0, vectorOut);

}

void SLAE::CalculateZ\_LUaster(double\* vectorOut)

{

SolveBackwardLU(auLU, diLU, tmp1, tmp1);

MatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(alLU, x0, tmp1);

SolveBackwardLU(alLU, tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(auLU, diLU, x0, vectorOut);

}

void SLAE::CalculateZ\_LUsq(double\* vectorOut)

{

SolveBackwardLU(auLU, diLU, tmp1, tmp1);

MatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(alLU, diLU, x0, tmp1);

SolveBackwardLU(alLU, diLU, tmp1, tmp1);

TransposedMatrixVectorMultiplication(tmp1, x0);

SolveForwardLU(auLU, diLU, x0, vectorOut);

}

double SLAE::CalculateRelativeDiscrepancyWithR(double norm)

{

return VectorNorm(r) / norm;

}

double SLAE::CalculateRelativeDiscrepancy(double norm)

{

MatrixVectorMultiplication(x, tmp1);

VectorSubtract(b, tmp1, tmp1);

return VectorNorm(tmp1) / norm;

}

void SLAE::AllocateMemory()

{

al = new double[nProfile]();

au = new double[nProfile]();

di = new double[n]();

ja = new int[nProfile]();

b = new double[n]();

x = new double[n]();

x0 = new double[n]();

r = new double[n]();

z = new double[n]();

tmp1 = new double[n]();

dP = new double[n]();

xtrue = new double[n]();

alLU = new double[nProfile]();

auLU = new double[nProfile]();

diLU = new double[n]();

}

void SLAE::ClearMemory()

{

delete[] al;

delete[] au;

delete[] di;

delete[] ja;

delete[] b;

delete[] x;

delete[] x0;

delete[] z;

delete[] tmp1;

}

PZ3.cpp

#include <iostream>

#include <chrono>

#include "slae.h"

using namespace std::chrono;

using namespace std;

//Не забыть вынести сделать функцию для подсчета относитльной невязки (F-Ax)

int main()

{

setlocale(LC\_ALL, "Russian");

int size;

SLAE slae;

FILE\* paramf, \* iaf, \* jaf, \* alf, \* auf, \* dif, \* bf;

FILE\* out;

fopen\_s(&out, "out.txt", "w");

double seconds;

int menu;

auto start\_time = steady\_clock::now();

auto end\_time = steady\_clock::now();

printf("Введите пункт меню:\n");

printf("1) МСГ для симметричной матрицы без обуславливания\n");

printf("2) МСГ для симметричной матрицы с диагональным обуславливанием\n");

printf("3) МСГ для симметричной матрицы с LU обуславливанием\n\n");

printf("4) МСГ для несимметричной матрицы без обуславливания \n");

printf("5) МСГ для несимметричной матрицы с диагональным обуславливанием \n");

printf("6) МСГ для несимметричной матрицы с LU обуславливанием \n");

printf("7) МСГ для несимметричной матрицы с LU\* обуславливанием \n");

printf("8) МСГ для несимметричной матрицы с LU(sq) обуславливанием \n\n");

printf("9) МСГ для матрицы Гильберта без обуславливания \n");

printf("10) МСГ для матрицы Гильберта с диагональным обуславливанием \n");

printf("11) МСГ для матрицы Гильберта с LU обуславливанием \n");

printf("12) МСГ для матрицы Гильберта с LU\* обуславливанием \n");

printf("13) МСГ для матрицы Гильберта с LU(sq) обуславливанием \n");

scanf\_s("%d", &menu);

setlocale(LC\_ALL, "US");

switch (menu)

{

case 1:

fopen\_s(&paramf, "param8.txt", "r");

fopen\_s(&iaf, "ia8.txt", "r");

fopen\_s(&jaf, "ja8.txt", "r");

fopen\_s(&alf, "al8.txt", "r");

fopen\_s(&auf, "au8.txt", "r");

fopen\_s(&dif, "di8.txt", "r");

fopen\_s(&bf, "b8.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

slae.OutputDense();

slae.MethodOfConjugateGradientsForSymMatrix();

slae.VectorOutputSolution(out);

break;

case 2:

fopen\_s(&paramf, "param4545.txt", "r");

fopen\_s(&iaf, "ia4545.txt", "r");

fopen\_s(&jaf, "ja4545.txt", "r");

fopen\_s(&alf, "al4545.txt", "r");

fopen\_s(&auf, "au4545.txt", "r");

fopen\_s(&dif, "di4545.txt", "r");

fopen\_s(&bf, "b4545.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

//slae.OutputDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForSymMatrixWithDiagP();

end\_time = steady\_clock::now();

cout << duration\_cast<microseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 3:

fopen\_s(&paramf, "param8.txt", "r");

fopen\_s(&iaf, "ia8.txt", "r");

fopen\_s(&jaf, "ja8.txt", "r");

fopen\_s(&alf, "al8.txt", "r");

fopen\_s(&auf, "au8.txt", "r");

fopen\_s(&dif, "di8.txt", "r");

fopen\_s(&bf, "b8.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

slae.OutputDense();

slae.CalculateLU();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForSymMatrixWithLuP();

end\_time = steady\_clock::now();

cout << duration\_cast<microseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 4:

fopen\_s(&paramf, "param12.txt", "r");

fopen\_s(&iaf, "ia12.txt", "r");

fopen\_s(&jaf, "ja12.txt", "r");

fopen\_s(&alf, "al12+.txt", "r");

fopen\_s(&auf, "au12+.txt", "r");

fopen\_s(&dif, "di12.txt", "r");

fopen\_s(&bf, "b12+.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

//slae.OutputDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrix();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 5:

fopen\_s(&paramf, "param12.txt", "r");

fopen\_s(&iaf, "ia12.txt", "r");

fopen\_s(&jaf, "ja12.txt", "r");

fopen\_s(&alf, "al12+.txt", "r");

fopen\_s(&auf, "au12+.txt", "r");

fopen\_s(&dif, "di12.txt", "r");

fopen\_s(&bf, "b12+.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

//slae.OutputDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithDiagP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 6:

fopen\_s(&paramf, "param12.txt", "r");

fopen\_s(&iaf, "ia12.txt", "r");

fopen\_s(&jaf, "ja12.txt", "r");

fopen\_s(&alf, "al12+.txt", "r");

fopen\_s(&auf, "au12+.txt", "r");

fopen\_s(&dif, "di12.txt", "r");

fopen\_s(&bf, "b12+.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

//slae.OutputDense();

slae.CalculateLU();

//slae.OutputLUDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithLuP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 7:

fopen\_s(&paramf, "param12.txt", "r");

fopen\_s(&iaf, "ia12.txt", "r");

fopen\_s(&jaf, "ja12.txt", "r");

fopen\_s(&alf, "al12+.txt", "r");

fopen\_s(&auf, "au12+.txt", "r");

fopen\_s(&dif, "di12.txt", "r");

fopen\_s(&bf, "b12+.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

//slae.OutputDense();

slae.CalculateLUaster();

//slae.OutputLUDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithLuAsterP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 8:

fopen\_s(&paramf, "param12.txt", "r");

fopen\_s(&iaf, "ia12.txt", "r");

fopen\_s(&jaf, "ja12.txt", "r");

fopen\_s(&alf, "al12+.txt", "r");

fopen\_s(&auf, "au12+.txt", "r");

fopen\_s(&dif, "di12.txt", "r");

fopen\_s(&bf, "b12+.txt", "r");

slae.Input(paramf, iaf, jaf, alf, auf, dif, bf);

//slae.OutputDense();

slae.CalculateLUsq();

//slae.OutputLUDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithLuSqP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 9:

slae.GenerateHilbertMatrix(10);

//slae.OutputDense();

//slae.OutputLUDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrix();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 10:

slae.GenerateHilbertMatrix(10);

//slae.OutputDense();

//slae.OutputLUDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithDiagP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 11:

slae.GenerateHilbertMatrix(8);

//slae.OutputDense();

//slae.OutputLUDense();

slae.CalculateLU();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithLuP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 12:

slae.GenerateHilbertMatrix(8);

slae.OutputDense();

slae.CalculateLUaster();

slae.OutputLUDense();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithLuAsterP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

case 13:

slae.GenerateHilbertMatrix(10);

//slae.OutputDense();

//slae.OutputLUDense();

slae.CalculateLUsq();

start\_time = steady\_clock::now();

slae.MethodOfConjugateGradientsForNonSymMatrixWithLuSqP();

end\_time = steady\_clock::now();

cout << duration\_cast<milliseconds>(end\_time - start\_time).count();

slae.VectorOutputSolution(out);

break;

default:

printf("Неправильный пункт меню!");

break;

}

}