МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

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

НГТУ НЭТИ

Кафедра прикладной математики

Практическая работа №3

по дисциплине «Численные методы»

ФПМИ

Группа	ПМ-92
Вариант	5
Студенты	Кутузов Иван
	Иванов Владислав
Преподаватель	Задорожный А. Г.
Дата	11.11.2021

Новосибирск

Цель работы

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

Вариант 5: Локально-оптимальная схема для несимметричной матрицы. Факторизация LU.

Анализ

Пусть имеется система линейных алгебраических уравнений:

$$Ax = F$$

Выбирается начальное приближение $x^{(0)} = (x_1^0, x_2^0, ..., x_n^0)$ (при отсутствии априорных данных для выбора приближения в качестве начального можно выбрать нулевой вектор).

Локально-оптимальная схема:
Предположим:

$$r^0 = f - Ax^0,$$

$$z^0 = r^0,$$

$$p^0 = Az^0.$$

Далее для $k = 1, 2, ...$ производятся следующие вычисления:

$$\alpha_k = \frac{\left(p^{k-1}, r^{k-1}\right)}{\left(p^{k-1}, p^{k-1}\right)},$$

$$x^k = x^{k-1} + \alpha_k z^{k-1},$$

$$r^k = r^{k-1} - \alpha_k p^{k-1},$$

$$\beta_k = -\frac{\left(p^{k-1}, Ar^k\right)}{\left(p^{k-1}, p^{k-1}\right)},$$

$$z^k = r^k + \beta_k z^{k-1},$$

$$p^k = Ar^k + \beta_k p^{k-1},$$

где $p^k = Az^k$ - вспомогательный вектор, который вычисляется рекуррентно.

Процесс заканчивается, если квадрат нормы невязки (r^k, r^k) достаточно мал.

Локально-оптимальная схема при использовании неполной факторизации:
Можно показать, что при использовании матриц неполной факторизации L и U локально-оптимальная схема преобразуется к следующему виду.
Полагаем:

$$\tilde{r}^0 = L^{-1}\left(f - Ax^0\right),$$

$$\hat{z}^0 = U^{-1}\tilde{r}^0, \qquad \hat{p}^0 = L^{-1}A\hat{z}^0.$$

Далее для $k = 1, 2, \ldots$ проводятся следующие вычисления:

$$\tilde{\alpha}_k = \frac{\left(\hat{p}^{k-1}, \tilde{r}^{k-1}\right)}{\left(\hat{p}^{k-1}, \hat{p}^{k-1}\right)},$$

$$x^k = x^{k-1} + \tilde{\alpha}_k \hat{z}^{k-1},$$

$$\tilde{r}^k = \tilde{r}^{k-1} - \tilde{\alpha}_k \hat{p}^{k-1},$$

$$\tilde{\beta}_k = -\frac{\left(\hat{p}^{k-1}, L^{-1}AU^{-1}\tilde{r}^k\right)}{\left(\hat{p}^{k-1}, \hat{p}^{k-1}\right)},$$

$$\hat{z}^k = U^{-1}\tilde{r}^k + \tilde{\beta}_k \hat{z}^{k-1},$$

$$\hat{p}^k = L^{-1}AU^{-1}\tilde{r}^k + \tilde{\beta}_k \hat{p}^{k-1}.$$

Процесс завершается, когда величина $\left(\tilde{r}^k, \tilde{r}^k\right)$ стала достаточно малой.

Разложение LU имеет следующий вид:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Ненулевые элементы матриц L и U вычисляются следующим образом:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \quad i = 1,..., n; \quad j = 1,..., i$$

$$u_{ij} = \frac{1}{l_{ii}} \left[a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right] \quad i = 1,..., n; \quad j = i + 1,..., n$$

Нулевые элементы остаются без изменения.

Прямой и обратный ход соответственно считаются по формулам:

$$y_i = F_i - \frac{1}{a_{ii}} \left[\sum_{j=1}^{i} a_{ij} F_j \right] \quad i = 1,..., n; \quad j = 1,..., i$$

$$x_j = y_j - \sum_{i=j}^{n} a_{ij} y_i \quad i = j,..., n; \quad j = 1,..., n$$

Исследования на матрицы с диагональным преобладанием

Матрица с диагональным преобладанием:

7	-1	0	0	0	-2	-3	0	0	0		1	-28
-2	9	-4	0	0	0	-1	-2	0	0		2	-19
0	0	7	-4	0	0	0	-2	-1	0		3	-20
0	0	-4	13	-3	0	0	0	-2	-4		4	-33
0	0	0	-3	6	-2	0	0	0	-1	*	5	-4
-1	0	0	0	-4	5	0	0	0	0	=	6	9
0	-4	0	0	0	0	5	-1	0	0		7	19
0	-1	-3	0	0	0	-2	7	-1	0		8	22
0	0	-3	-2	0	0	0	-1	6	0		9	29
0	0	0	-2	-1	0	0	0	-2	5		10	19

Сравнение методов:

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	100000	66.04300	5.0013876308192e-03
ЛОС (диагональная факторизация)	100000	81.40420	1.6815416169171e-03
ЛОС (LU факторизация)	100000	99.64210	4.0701208418130e-07
Якоби	55697	18.24459	9.9685776700715e-14
Гаусс-Зейдель	3030	0.98500	8.5343320793939e-14

Результат:

ЛОС	**ЛОС** (диагональная факторизация)	**ЛОС** (**LU**факторизация)
-3.847979455140894	-3.461000899186617	0.999328537007728
-3.600730307776493	-3.162306369419506	1.999222296801105
-2.860142107661058	-2.403171532064211	2.999188272733557
-1.873292014957248	-1.416791659361339	3.999186943276795
-0.832661490970150	-0.365395004201319	4.999194853166109
0.336329218224806	0.801213662527984	5.999216554745624
1.349224994743832	1.794806330914686	6.999217327575082
2.225765780108788	2.678767792705368	7.999194985902064
3.132451699218045	3.593081419527142	8.999185614197524
4.109335465972014	4.577384047930832	9.999186570923632

Матрица с обратными знаками внедиагональных элементов:

7	1	0	0	0	2	3	0	0	0		1		42
2	9	4	0	0	0	1	2	0	0		2		55
0	0	7	4	0	0	0	2	1	0		3		62
0	0	4	13	3	0	0	0	2	4		4		137
0	0	0	3	6	2	0	0	0	1	*	5	=	64
1	0	0	0	4	5	0	0	0	0		6		51
0	4	0	0	0	0	5	1	0	0		7		51
0	1	3	0	0	0	2	7	1	0		8		90
0	0	3	2	0	0	0	1	6	0		9		79
0	0	0	2	1	0	0	0	2	5		10		81

Сравнение методов:

Метод	**Количество итераций**	**Время решения, мс**	**Относительная невязка**
ЛОС	57	0.04220	8.4613984302916e-14
ЛОС (диагональная факторизация)	48	0.03910	9.2638836593332e-14
ЛОС (LU факторизация)	12	0.01290	5.1813885187253e-14
Якоби	665	0.38460	7.9086358981517e-14
Гаусс-Зейдель	180	0.05740	4.1199755415227e-14

Результат:

ЛОС	**ЛОС** (диагональная факторизация)	**ЛОС** (LUфакторизация)
0.999999999998167	0.999999999994237	1.000000000000636
2.000000000001192	2.000000000007540	1.999999999994188
2.999999999996600	2.999999999988840	3.000000000009308
4.000000000001672	4.000000000008924	3.999999999992060
4.999999999996187	4.999999999987640	5.000000000006404
6.000000000004069	6.000000000014064	5.999999999993482
7.000000000000274	6.999999999991365	7.000000000009450
8.000000000002217	8.000000000005631	7.999999999992525
9.000000000000085	9.000000000002158	9.000000000002149
10.000000000000854	9.999999999995477	10.000000000004176

Исследование на матрицах Гильберта различных размерностей

Матрица Гильберта размерностью 5:

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	6	0.00300	7.9064701671646e-14
ЛОС (диагональная факторизация)	6	0.00370	1.2542187999324e-14
ЛОС (LU факторизация)	1	0.00110	1.9376988400218e-16

Матрица Гильберта размерностью 8:

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	12	0.00920	1.6332622704326e-14
ЛОС (диагональная факторизация)	11	0.02220	8.2325297732638e-14
ЛОС (LU факторизация)	1	0.00200	1.7786135499230e-16

Матрица Гильберта размерностью 10:

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	16	0.01500	1.1189456666158e-14
ЛОС (диагональная факторизация)	15	0.01490	1.9018201107824e-14
ЛОС (LU факторизация)	2	0.00360	2.2916680416514e-17

Исследования на матрицах большой размерности

***0945:**

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	561	47.41620	9.9266300538289e-23
ЛОС (диагональная факторизация)	77	7.20250	9.7085443060247e-23
ЛОС (LU факторизация)	10	1.84050	4.3634913425971e-23

***4545:**

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	2648	1112.00860	9.8965703547765e-23
ЛОС (диагональная факторизация)	283	128.34560	9.7885451816704e-23
ЛОС (LU факторизация)	10	9.35240	2.3623754779641e-23

Вывод

Как видно из первого теста, для первой матрицы с диагональным преобладанием локально-оптимальная схема была хуже (происходит аварийный выход по максимальному количеству итераций), чем метод Якоби и Гаусса-Зейделя.
Однако для второй матрицы с обратным знаком внедиагональных элементов локально-оптимальная схема превосходит метод Якоби и Гаусса-Зейделя, как по количеству итераций, так и по времени решения.

Использование факторизации исходной матрицы повышает точность получаемого значения и уменьшает количество итераций, необходимых для нахождения решения, но увеличивается сложность каждой отдельной итерации. Факторизацию тем эффективнее, чем хуже обусловленность матрицы СЛАУ. За счет предобуславливания исходной матрицы, увеличивается скорость сходимости, благодаря чему можно получить выигрыш по времени, что хорошо видно в тестах на матрицах большой размерности и на матрицах Гильберта.

Приложение

Диагональная матрица -> результат за 1 итерацию:

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС (диагональная факторизация)	1	0.00290	9.5083839368171e-17

Начальное приближение совпадает с искомым решением -> результат за 0 итераций:

Метод	Количество итераций	Время решения, мс	Относительная невязка
ЛОС	0	0.00210	-
ЛОС (диагональная факторизация)	0	0.00180	-
ЛОС (LU факторизация)	0	0.00190	-

Текст программы

io.h

```
#pragma once
#include "common.h"
#include "AlgebraicStructures.h"

double* ReadVectorReal(const char* filePath);
int* ReadVectorInt(const char* filePath);
int* ReadVectorInt_ForTestFiles(const char* filePath);
void ReadParameters(int* maxiter, real* eps, int* n, int* nonDiagonalElements,
const char* filePath);

void WriteVectorReal(double* vector, int n, const char* filePath);

void WriteTable(int iterations, double time, double relativeDiscrepancy, const
char* filePath);
```

io.c

```
#include "io.h"
#include <stdio.h>
#include <corecrt_malloc.h>

double* ReadVectorReal(const char* filePath)
{
	double* vector = NULL;
	FILE* stream = NULL;

	fopen_s(&stream, filePath, "r");

	if (stream != NULL)
	{
		int n;
		fscanf_s(stream, "%d", &n);
		vector = malloc(sizeof(double) * n);
		for (int i = 0; i < n; i++)
		{
			fscanf_s(stream, "%lf", (vector + i));
		}

		fclose(stream);
	}

	return vector;
}

int* ReadVectorInt(const char* filePath)
{
	int* vector = NULL;
	FILE* stream = NULL;

	fopen_s(&stream, filePath, "r");

	if (stream != NULL)
	{
		int n;
		fscanf_s(stream, "%d", &n);
		vector = malloc(sizeof(int) * n);
		for (int i = 0; i < n; i++)
		{
			fscanf_s(stream, "%d", (vector + i));
		}
```

```
		fclose(stream);
	}

	return vector;
}

int* ReadVectorInt_ForTestFiles(const char* filePath)
{
	int* vector = NULL;
	FILE* stream = NULL;

	fopen_s(&stream, filePath, "r");

	if (stream != NULL)
	{
		int n;
		fscanf_s(stream, "%d", &n);
		vector = malloc(sizeof(int) * n);
		for (int i = 0; i < n; i++)
		{
			fscanf_s(stream, "%d", (vector + i));
			*(vector + i) = *(vector + i) - 1;
		}

		fclose(stream);
	}

	return vector;
}

void ReadParameters(int* maxiter, real* eps, int* n, int* nonDiagonalElements,
const char* filePath)
{
	FILE* stream = NULL;

	fopen_s(&stream, filePath, "r");

	if (stream != NULL)
	{
		fscanf_s(stream, "%d", n);
		fscanf_s(stream, "%d", nonDiagonalElements);
		fscanf_s(stream, "%d", maxiter);
		fscanf_s(stream, "%lf", eps);

		fclose(stream);
	}
}

void WriteVectorReal(double* vector, int n, const char* filePath)
{
	FILE* stream = NULL;

	fopen_s(&stream, filePath, "a+");

	if (stream != NULL)
	{
		fprintf_s(stream, "--------------\n");
		for (int i = 0; i < n; i++)
		{
			fprintf_s(stream, "%.15lf\n", vector[i]);
		}
		fprintf_s(stream, "--------------\n");

		fclose(stream);
	}
}
```

```
void WriteTable(int iterations, double time, double relativeDiscrepancy, const
char* filePath)
{
	FILE* stream = NULL;

	fopen_s(&stream, filePath, "a+");

	if (stream != NULL)
	{
		fprintf_s(stream, "--------------\n");
		fprintf_s(stream, "iterations: %d\n", iterations);
		fprintf_s(stream, "time: %.7lf ms\n", time);
		fprintf_s(stream, "time: %.13e\n", relativeDiscrepancy);
		fprintf_s(stream, "--------------\n");

		fclose(stream);
	}
}
```

Timer.h

```
#pragma once
#include <Windows.h>

LARGE_INTEGER StartTimer();
LARGE_INTEGER StopTimer();
double ComputeElapsedTime(LARGE_INTEGER start, LARGE_INTEGER stop);
```

Timer.c

```
#include "Timer.h"

LARGE_INTEGER StartTimer()
{
	LARGE_INTEGER time;
	QueryPerformanceCounter(&time);

	return time;
}

LARGE_INTEGER StopTimer()
{
	LARGE_INTEGER time;
	QueryPerformanceCounter(&time);

	return time;
}

double ComputeElapsedTime(LARGE_INTEGER start, LARGE_INTEGER stop)
{
	LARGE_INTEGER frequency;

	QueryPerformanceFrequency(&frequency);

	return (stop.QuadPart - start.QuadPart) * 1000.0 / frequency.QuadPart;
}
```

source.c

```c
#include "io.h"
#include <stdio.h>
#include <corecrt_malloc.h>
#include <math.h>
#include "Timer.h"

//matrix(not factorized)
double* di;
double* al;
double* au;

int* ia;
int* ja;
//

//diagonal factorization
double* dif;// di^(-1)
//

//LU factorization
double* alf;
double* auf;
//

//vectors
double* r;
double* z;
double* p;
double* x;
double* F;
double* tmp;
//

//parameters
int n;
int nonDiagonalElements;
int maxiter;
double eps;
//

//
double FNorm;
double elapsedTime;
double relativeDiscrepancy;
//

void GenerateHilbertMatrix()
{
	di = malloc(sizeof(double) * n);
	au = malloc(sizeof(double) * nonDiagonalElements);
	al = malloc(sizeof(double) * nonDiagonalElements);
	ia = malloc(sizeof(int) * (n + 1));
	ja = malloc(sizeof(int) * nonDiagonalElements);

	ia[0] = 0;
	ia[1] = 0;
	for (int i = 1; i < n; i++)
	{
		ia[i + 1] = ia[i] + i;
	}

	for (int i = 0; i < n; i++)
	{
		int cur_i = ia[i];
		int next_i = ia[i + 1];
```

```
                int j = 0;
                for (int k = cur_i; k < next_i; k++, j++)
                {
                        ja[k] = j;
                        al[k] = 1.0 / (double)(i + j + 1);
                        au[k] = 1.0 / (double)(i + j + 1);
                }

                di[i] = 1.0 / (double)(i + j + 1);
        }
}

void MultiplyMatrixByVector(double* vector, double* result)
{
        double* tempVector = malloc(sizeof(double) * n);

        for (int i = 0; i < n; i++)
        {
                tempVector[i] = 0.0;
        }

        for (int i = 0; i < n; i++)
        {
                tempVector[i] = di[i] * vector[i];

                int cur_i = ia[i];
                int next_i = ia[i + 1];

                for (int k = cur_i; k < next_i; k++)
                {
                        tempVector[i] += al[k] * vector[ja[k]];
                        tempVector[ja[k]] += au[k] * vector[i];
                }
        }

        for (int i = 0; i < n; i++)
        {
                result[i] = tempVector[i];
        }

        free(tempVector);
}

double MultiplyScalar(double* first, double* second)
{
        double sum = 0.0;
        for (int i = 0; i < n; i++)
        {
                sum += first[i] * second[i];
        }
        return sum;
}

double CalculateEuqlideanNorm(double* vector)
{
        double sum = 0.0;
        for (int i = 0; i < n; i++)
        {
                sum += vector[i] * vector[i];
        }
        return sqrt(sum);
}

double CalculateRelativeDiscrepancy()
{
        //return MultiplyScalar(r, r);
        return CalculateEuqlideanNorm(r) / FNorm;
```

```
}

void CalculateXk(double alpha)
{
	for (int i = 0; i < n; i++)
	{
		x[i] += alpha * z[i];
	}
}

void CalculateRk(double alpha)
{
	for (int i = 0; i < n; i++)
	{
		r[i] -= alpha * p[i];
	}
}

void CalculateZk(double beta)
{
	for (int i = 0; i < n; i++)
	{
		z[i] = r[i] + beta * z[i];
	}
}

void CalculateZk_WithFactorization(double beta)
{
	for (int i = 0; i < n; i++)
	{
		z[i] = tmp[i] + beta * z[i];
	}
}

void CalculatePk(double beta)
{
	for (int i = 0; i < n; i++)
	{
		p[i] = tmp[i] + beta * p[i];
	}
}

int SolveByLos()
{
	for (int i = 0; i < n; i++)
	{
		x[i] = 0.0;
		r[i] = F[i];
		z[i] = r[i];
	}
	MultiplyMatrixByVector(z, p);

	LARGE_INTEGER start = StartTimer();

	double pp = MultiplyScalar(p, p);
	double alpha = MultiplyScalar(p, r) / pp;

	int i = 0;
	for (i; i < maxiter && CalculateRelativeDiscrepancy() > eps; i++)
	{
		CalculateXk(alpha);
		 CalculateRk(alpha);

		 MultiplyMatrixByVector(r, tmp);//tmp is Ar
		double beta = -(MultiplyScalar(p, tmp)) / pp;

		CalculateZk(beta);
```

```
                CalculatePk(beta);

                pp = MultiplyScalar(p, p);
                alpha = MultiplyScalar(p, r) / pp;
        }

        LARGE_INTEGER stop = StopTimer();
        elapsedTime = ComputeElapsedTime(start, stop);

        relativeDiscrepancy = CalculateRelativeDiscrepancy();
        return i;
}

void DiagonalFactorization()
{
        for (int i = 0; i < n; i++)
        {
                dif[i] = 1.0 / sqrt(di[i]);
        }
}

void MultiplyDiagonalByVector(double* vector, double* result)
{
        for (int i = 0; i < n; i++)
        {
                result[i] = dif[i] * vector[i];
        }
}

int SolveByLos_WithDiagonalFactorization()
{
        dif = malloc(sizeof(double) * n);

        DiagonalFactorization();

        for (int i = 0; i < n; i++)
        {
                x[i] = 0.0;
                r[i] = F[i];//r = F - Ax
        }
        MultiplyDiagonalByVector(r, r);//r = D'(F - Ax)
        MultiplyDiagonalByVector(r, z);//z = D'r
        MultiplyMatrixByVector(z, p);//p = Az
        MultiplyDiagonalByVector(p, p);// p = D'Az

        LARGE_INTEGER start = StartTimer();

        double pp = MultiplyScalar(p, p);
        double alpha = MultiplyScalar(p, r) / pp;

        int i = 0;
        for (i; i < maxiter && CalculateRelativeDiscrepancy() > eps; i++)
        {
                CalculateXk(alpha);
                CalculateRk(alpha);

                 MultiplyDiagonalByVector(r, tmp);
                MultiplyMatrixByVector(tmp, tmp);
                MultiplyDiagonalByVector(tmp, tmp);
                double beta = -(MultiplyScalar(p, tmp) / pp);

                CalculatePk(beta);
                MultiplyDiagonalByVector(r, tmp);
                CalculateZk_WithFactorization(beta);

                pp = MultiplyScalar(p, p);
                alpha = MultiplyScalar(p, r) / pp;
```

```
	}

	LARGE_INTEGER stop = StopTimer();
	elapsedTime = ComputeElapsedTime(start, stop);

	relativeDiscrepancy = CalculateRelativeDiscrepancy();

	free(dif);

	return i;
}

void LuFactorization()
{
	for (int i = 0; i < n; i++)
	{
		int cur_i = ia[i];
		int next_i = ia[i + 1];

		double sumd = 0.0;
		for (int k = cur_i; k < next_i; k++)
		{
			int j = ja[k];//текущий столбец
			int ki = cur_i;
			int kj = ia[j];
			int j1 = ia[j + 1];

			double suml = 0.0;
			double sumu = 0.0;

			while (ki < k && kj < j1)
			{
				if (ja[ki] == ja[kj])
				{
					suml += alf[ki] * auf[kj];
					sumu += auf[ki] * alf[kj];
					ki++;
					kj++;
				}
				else
				{
					if (ja[ki] > ja[kj])
					{
						kj++;
					}
					else
					{
						ki++;
					}
				}
			}

			alf[k] = al[k] - suml;
			auf[k] = (au[k] - sumu) / dif[j];

			sumd += alf[k] * auf[k];
		}

		dif[i] = di[i] - sumd;
	}
}

void ForwardStep(double* vector, double* result)
{
	for (int i = 0; i < n; i++)
	{
		double sum = 0.0;
```

```
                int cur_i = ia[i];
                int next_i = ia[i + 1];

                for (int j = cur_i; j < next_i; j++)
                {
                        sum += alf[j] * result[ja[j]];
                }
                result[i] = (vector[i] - sum) / dif[i];
        }
}

void ReverseStep(double* vector, double* result)
{
        for (int i = 0; i < n; i++)
        {
                result[i] = vector[i];
        }

        for (int i = n - 1; i >= 0; i--)
        {
                int cur_i = ia[i];
                int next_i = ia[i + 1];
                for (int j = cur_i; j < next_i; j++)
                {
                        result[ja[j]] -= auf[j] * result[i];
                }
        }
}

int SolveByLos_WithLuFactorization()
{
        alf = malloc(sizeof(double) * nonDiagonalElements);
        dif = malloc(sizeof(double) * n);
        auf = malloc(sizeof(double) * nonDiagonalElements);

        LuFactorization();

        for (int i = 0; i < n; i++)
        {
                x[i] = 0.0;
                r[i] = F[i];//r = F - Ax
        }
        ForwardStep(r, r);
        ReverseStep(r, z);
        MultiplyMatrixByVector(z, p);
        ForwardStep(p, p);

        LARGE_INTEGER start = StartTimer();

        double pp = MultiplyScalar(p, p);
        double alpha = MultiplyScalar(p, r) / pp;

        int i = 0;
        for (i; i < maxiter && CalculateRelativeDiscrepancy() > eps; i++)
        {
                CalculateXk(alpha);
                CalculateRk(alpha);

                ReverseStep(r, tmp);
                MultiplyMatrixByVector(tmp, tmp);
                ForwardStep(tmp, tmp);
                double beta = -(MultiplyScalar(p, tmp) / pp);

                CalculatePk(beta);
                ReverseStep(r, tmp);
                CalculateZk_WithFactorization(beta);
```

```
            pp = MultiplyScalar(p, p);
            alpha = MultiplyScalar(p, r) / pp;
        }

        LARGE_INTEGER stop = StopTimer();
        elapsedTime = ComputeElapsedTime(start, stop);

        relativeDiscrepancy = CalculateRelativeDiscrepancy();

        free(dif);
        free(alf);
        free(auf);

        return i;
}

int main()
{
        ReadParameters(&maxiter, &eps, &n, &nonDiagonalElements,
"Test\\Hilbert10\\kuslau.txt");

        //di = ReadVectorReal("Test\\0945\\di.txt");
        //al = ReadVectorReal("Test\\0945\\ggl.txt");
        //au = ReadVectorReal("Test\\0945\\ggu.txt");
        //ia = ReadVectorInt("Test\\MatrixB\\ig.txt");
        //ja = ReadVectorInt("Test\\MatrixB\\jg.txt");
        //ia = ReadVectorInt_ForTestFiles("Test\\0945\\ig.txt");
        //ja = ReadVectorInt_ForTestFiles("Test\\0945\\jg.txt");

        GenerateHilbertMatrix();

        F = ReadVectorReal("Test\\Hilbert10\\pr.txt");
        FNorm = CalculateEuqlideanNorm(F);

        x = malloc(sizeof(double) * n);
        r = malloc(sizeof(double) * n);
        z = malloc(sizeof(double) * n);
        p = malloc(sizeof(double) * n);
        tmp = malloc(sizeof(double) * n);

        //int iterations = SolveByLos();
        //int iterations = SolveByLos_WithDiagonalFactorization();
        int iterations = SolveByLos_WithLuFactorization();

        WriteVectorReal(x, n, "Test\\Hilbert10\\resultVector.txt");
        WriteTable(iterations, elapsedTime, relativeDiscrepancy,
"Test\\Hilbert10\\table.txt");

        free(di);
        free(al);
        free(au);
        free(ia);
        free(ja);
        free(x);
        free(r);
        free(z);
        free(p);
        free(tmp);
        return 0;}
```