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 №1

по дисциплине «Численные методы»

ФПМИ

Группа	ПМ-92
Вариант	7
Студенты	Кутузов Иван
	Иванов Владислав
Преподаватель	Задорожный А. Г.
Дата	08.10.2021

Новосибирск

Цель работы

Разработать решение СЛАУ прямым методом с хранением матрицы в профильном формате. Исследовать накопление погрешности и ее зависимость от числа обусловленности. Сравнить метод по точности получаемого решения и количеству действий с методом Гаусса.

Анализ

Пусть имеется система линейных алгебраических уравнений:

$$Ax = F$$

Представим матрицу A в виде произведения нижнетреугольной и верхнетреугольной матриц LU. Тогда решение СЛАУ сводится к решению двух систем с треугольными матрицами:

1. $LUx = F$.
2. $Ly = F,\ Ux = y$

LU разложение существует только в том случае, когда матрица A обратима, а все ведущие(угловые) главные миноры матрицы A невырождены.

Разложение LU:

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

Где элементы матрицы L и U вычисляются следующим образом:

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

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

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

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

Профильный формат матрицы предполагает хранить по возможности только ненулевые элементы и имеет следующий вид:

$$
\begin{bmatrix}
a_{11} & & & & & & & & \\
 & a_{22} & a_{23} & a_{24} & & & & 0 & \\
 & a_{32} & a_{33} & 0 & a_{35} & a_{36} & & & \\
 & a_{42} & 0 & a_{44} & a_{45} & 0 & a_{47} & & \\
 & & a_{53} & a_{54} & a_{55} & a_{56} & 0 & a_{58} & a_{59} \\
 & & a_{63} & 0 & a_{65} & a_{66} & 0 & a_{68} & 0 \\
 & 0 & & a_{74} & 0 & 0 & a_{77} & 0 & a_{79} \\
 & & & & a_{85} & a_{86} & 0 & a_{88} & 0 \\
 & & & & a_{95} & 0 & a_{97} & 0 & a_{99}
\end{bmatrix}
$$

Текст программы

Программа была разбита на следующие модули:

common.h — содержит макрос, позволяющий быстро изменить точность вычисления (double или float).

io.h — содержит процедуры для чтения и записи данных в файл.

LinearAlgebra.h — содержит определения структур данных векторов и матриц, процедуры их заполнения(генерация матрицы Гильберта), а также матрично-векторное произведение.

LowUpDecomposition.h — содержит процедуры для разложения матрицы.

SystemOfEquation.h — содержит процедуры для решения СЛАУ(методом разложения и методом Гаусса с ведущим элементом).

Test.h — содержит процедуры, позволяющие провести исследования различных методов решений СЛАУ(полное решение методом разложения, методом Гаусса и решение для матриц Гильберта).

common.h

```c
#ifndef FLOAT
	typedef float real;
	typedef float real_sum;
#endif
```

io.h

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

#define FLOAT

void ReadVectorReal(struct VectorReal* vector, const char* filePath);

void ReadVectorInt(struct VectorInt* vector, const char* filePath);

void ReadDenseMatrix(struct Matrix* matrix, const char* filePath);

void WriteVectorReal(struct VectorReal* vector, const char* filePath);
```

io.c

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

void ReadVectorReal(struct VectorReal* vector, const char* filePath)
{
	FILE* stream = NULL;

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

	if (stream != NULL)
	{
		fscanf_s(stream, "%d", &vector->size);
		vector->data = malloc(sizeof(real) * vector->size);
		for (int i = 0; i < vector->size; i++)
		{
			fscanf_s(stream, "%f", (vector->data + i));
		}

		fclose(stream);
	}
}

void ReadVectorInt(struct VectorInt* vector, const char* filePath)
{
	FILE* stream = NULL;

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

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

		fclose(stream);
	}
}

void ReadDenseMatrix(struct Matrix* matrix, const char* filePath)
{
	FILE* stream = NULL;

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

	if (stream != NULL)
	{
		fscanf_s(stream, "%d", &matrix->size);
		matrix->data = malloc(sizeof(real) * matrix->size);
		for (int i = 0; i < matrix->size; i++)
		{
			matrix->data[i] = malloc(sizeof(real) * matrix->size);
			for (int j = 0; j < matrix->size; j++)
			{
				fscanf_s(stream, "%f", &matrix->data[i][j]);
			}
		}

		fclose(stream);
	}
```

```
}

void WriteVectorReal(struct VectorReal* vector, const char* filePath)
{
	FILE* stream = NULL;

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

	if (stream != NULL)
	{
		fprintf_s(stream, "---------------\n");
		for (int i = 0; i < vector->size; i++)
		{
			fprintf_s(stream, "%.8f\n", vector->data[i]);
		}
		fprintf_s(stream, "---------------\n");

		fclose(stream);
	}
}
```

LinearAlgebra.h

```
#include "common.h"

struct VectorReal
{
	int size;
	real* data;
};

struct VectorInt
{
	int size;
	int* data;
};

struct VectorRealSum
{
	int size;
	real_sum* data;
};

struct ProfileMatrix
{
	struct VectorReal* diagonal;
	struct VectorReal* upperElements;
	struct VectorReal* lowerElements;
	struct VectorInt* indexArray;
};

struct Matrix
{
	int size;
	real** data;
};

struct VectorReal* CreateVectorReal();
struct VectorRealSum* CreateVectorRealSum();
struct VectorInt* CreateVectorInt();
struct ProfileMatrix* CreateProfileMatrix();
struct Matrix* CreateDenseMatrix();

void ClearVectorReal(struct VectorReal* vector);
```

```
void ClearVectorRealSum(struct VectorRealSum* vector);
void ClearVectorInt(struct VectorInt* vector);
void ClearProfileMatrix(struct ProfileMatrix* matrix);
void ClearDenseMatrix(struct Matrix* matrix);

void GenerateHilbertMatrix(struct Matrix* matrix, int size);
void GenerateVector(struct VectorReal* vector, int size);

void MatrixVectorMultiply_ProfileMatrix(struct ProfileMatrix* matrix, struct
VectorReal* vector);
void MatrixVectorMultiply_DenseMatrix(struct Matrix* matrix, struct VectorReal*
vector);
```

LinearAlgebra.c

```
#include "LinearAlgebra.h"
#include <stddef.h>
#include <math.h>

struct VectorInt* CreateVectorInt()
{
	struct VectorInt* vector = malloc(sizeof(struct VectorInt));
	vector->size = 0;

	return vector;
}

struct VectorReal* CreateVectorReal()
{
	struct VectorReal* vector = malloc(sizeof(struct VectorReal));
	vector->size = 0;

	return vector;
}

struct VectorRealSum* CreateVectorRealSum()
{
	struct VectorRealSum* vector = malloc(sizeof(struct VectorRealSum));
	vector->size = 0;

	return vector;
}

struct ProfileMatrix* CreateProfileMatrix()
{
	struct ProfileMatrix* matrix = malloc(sizeof(struct ProfileMatrix));

	matrix->diagonal = CreateVectorReal();
	matrix->lowerElements = CreateVectorReal();
	matrix->upperElements = CreateVectorReal();
	matrix->indexArray = CreateVectorInt();

	return matrix;
}

struct Matrix* CreateDenseMatrix()
{
	struct Matrix* matrix = malloc(sizeof(struct Matrix));
	matrix->size = 0;

	return matrix;
}

void ClearVectorReal(struct VectorReal* vector)
```

```
{
	free(vector->data);
	vector->data = NULL;
	free(vector);
	vector = NULL;
}

void ClearVectorRealSum(struct VectorRealSum* vector)
{
	free(vector->data);
	vector->data = NULL;
	free(vector);
	vector = NULL;
}

void ClearVectorInt(struct VectorInt* vector)
{
	free(vector->data);
	vector->data = NULL;
	free(vector);
	vector = NULL;
}

void ClearProfileMatrix(struct ProfileMatrix* matrix)
{
	ClearVectorReal(matrix->diagonal);
	ClearVectorReal(matrix->upperElements);
	ClearVectorReal(matrix->lowerElements);
	ClearVectorInt(matrix->indexArray);
	free(matrix);
	matrix = NULL;
}

void ClearDenseMatrix(struct Matrix* matrix)
{
	for (int i = 0; i < matrix->size; i++)
	{
		free(matrix->data[i]);
		matrix->data[i] = NULL;
	}

	free(matrix);
	matrix = NULL;
}

void GenerateHilbertMatrix(struct Matrix* matrix, int size)
{
	matrix->size = size;
	matrix->data = (real**)malloc(sizeof(real*) * size);

	for (int i = 0; i < size; i++)
	{
		matrix->data[i] = (real*)malloc(sizeof(real) * size);

		for (int j = 0; j < size; j++)
		{
			matrix->data[i][j] = 1.0 / (double)(i + j + 1);
		}
	}
}

void GenerateVector(struct VectorReal* vector, int size)
{
	vector->size = size;
	vector->data = malloc(sizeof(real) * size);
```

```
	for (int i = 0; i < size; i++)
	{
		vector->data[i] = (real)(i + 1);
	}
}


void MatrixVectorMultiply_ProfileMatrix(struct ProfileMatrix* matrix, struct
VectorReal* vector)
{
	struct VectorRealSum* vectorTemp = NULL;
	vectorTemp = CreateVectorRealSum();
	vectorTemp->size = vector->size;
	vectorTemp->data = malloc(sizeof(real_sum) * vectorTemp->size);

	for (int i = 0; i < vector->size; i++)
	{
		vectorTemp->data[i] = 0.0;

		int indexOfFirstElement = matrix->indexArray->data[i];
		int nextLineIndex = matrix->indexArray->data[i + 1];
		int j = i - (nextLineIndex - indexOfFirstElement);

		for (int k = indexOfFirstElement; k < nextLineIndex; k++, j++)
		{
			vectorTemp->data[i] += matrix->lowerElements->data[k] *
vector->data[j];
			vectorTemp->data[j] += matrix->upperElements->data[k] *
vector->data[i];
		}

		vectorTemp->data[i] += matrix->diagonal->data[i] * vector->data[i];
	}

	for (int i = 0; i < vector->size; i++)
	{
		vector->data[i] = vectorTemp->data[i];
	}

	ClearVectorRealSum(vectorTemp);
}

void MatrixVectorMultiply_DenseMatrix(struct Matrix* matrix, struct VectorReal*
vector)
{
	struct VectorReal* vectorTemp = NULL;
	vectorTemp = CreateVectorReal();
	vectorTemp->size = vector->size;
	vectorTemp->data = malloc(sizeof(real) * vectorTemp->size);

	for (int i = 0; i < vector->size; i++)
	{
		vectorTemp->data[i] = 0;

		for (int j = 0; j < vector->size; j++)
		{
			vectorTemp->data[i] += matrix->data[i][j] * vector->data[j];
		}
	}

	for (int i = 0; i < vector->size; i++)
	{
		vector->data[i] = vectorTemp->data[i];
	}
```

```
    ClearVectorReal(vectorTemp);
}
```

LowUpDecomposition.h

```
#include "LinearAlgebra.h"

void DecomposeProfileMatrix(struct ProfileMatrix* matrix);

void DecomposeDenseMatrix(struct Matrix* matrix);
```

LowUpDecomposition.c

```
#include "LowUpDecomposition.h"
#include <math.h>

void DecomposeProfileMatrix(struct ProfileMatrix* matrix)
{
	int* ia = matrix->indexArray->data;
	int n = matrix->diagonal->size;

	for (int i = 1; i < n; i++)
	{
		int indexOfFirstElement = *(ia + i); //ia[i]
		int nextLineIndex = *(ia + i + 1); //ia[i + 1]
		int currentFirst = i - (nextLineIndex - indexOfFirstElement);
		int j = currentFirst;// индекс столбца первого хранящегося элемента
в плотном виде

		real_sum diagonalSum = 0.0;

		for (int k = indexOfFirstElement; k < nextLineIndex; k++, j++)
		{
			int previousLineIndex = *(ia + j);//ia[j]
			int previousFirst = j - (*(ia + j + 1) - *(ia + j));
			int current = indexOfFirstElement;
			int previous = previousLineIndex;

			int difference = previousFirst - currentFirst;

			if (difference < 0)//если в текущей строке меньше элементов
			{
				previous -= difference;
			}
			else
			{
				current += difference;
			}

			real_sum lowerSum = 0.0;
			real_sum upperSum = 0.0;
```

```
			for (current; current < k; current++, previous++)//количество итераций = min(кол-во элементов в текущей строке; кол-во элементов в предыдущей строке)
			{
				lowerSum += matrix->lowerElements->data[current] * 
matrix->upperElements->data[previous];//просто по формуле считаем
				upperSum += matrix->upperElements->data[current] * 
matrix->lowerElements->data[previous];
			}

			matrix->lowerElements->data[k] -= lowerSum;
			//upperSum /= matrix->diagonal->data[j];

			matrix->upperElements->data[k] = 
(matrix->upperElements->data[k] - upperSum) / matrix->diagonal->data[j];

			diagonalSum += matrix->lowerElements->data[k] * 
matrix->upperElements->data[k];//diagonal
		}

		matrix->diagonal->data[i] -= diagonalSum;
	}
}

void DecomposeDenseMatrix(struct Matrix* matrix)
{
	for (int i = 0; i < matrix->size; i++)
	{
		real_sum sum = 0.0;
		for (int k = 0; k < i; k++)
		{
			sum += matrix->data[i][k] * matrix->data[k][i];
		}
		matrix->data[i][i] -= sum;

		for (int j = i + 1; j < matrix->size; j++)
		{
			real_sum lowerSum = 0.0;
			real_sum upperSum = 0.0;
			for (int k = 0; k < i; k++)
			{
				lowerSum += matrix->data[j][k] * matrix->data[k][i];
				upperSum += matrix->data[i][k] * matrix->data[k][j];
			}

			matrix->data[j][i] -= lowerSum;

			matrix->data[i][j] = (matrix->data[i][j] - upperSum) / 
matrix->data[i][i];
		}
	}
}
```

SystemOfEquation.h

```c
#include "LinearAlgebra.h"
#include "common.h"

void SolveByLowUpDecomposition_ProfileMatrix(struct ProfileMatrix* matrix, struct
VectorReal* vector);

void SolveByLowUpDecomposition_DenseMatrix(struct Matrix* matrix, struct
VectorReal* vector);

void SolveByGauss(struct Matrix* matrix, struct VectorReal* vector);
```

SystemOfEquation.c

```c
void SolveByLowUpDecomposition_ProfileMatrix(struct ProfileMatrix* matrix, struct
VectorReal* vector)
{
	int i;
	int n = vector->size;
	int* ia = matrix->indexArray->data;
	real* al = matrix->lowerElements->data;
	real* di = matrix->diagonal->data;

	//Ly = b
	for (i = 0; i < n; i++, ia++, di++)
	{
		int indexOfFirstElement = *(ia);
		int nextLineIndex = *(ia + 1);
		int j = i - (nextLineIndex - indexOfFirstElement);
		real_sum sum = 0.0;

		for (int k = indexOfFirstElement; k < nextLineIndex; k++, j++, al++)
		{
			sum += *(al) * vector->data[j];
		}

		vector->data[i] = (vector->data[i] - sum) / *(di);
	}
	//

	//Ux = y
	ia -= 1;
	for (i -= 1; i >= 0; i--, ia--)
	{
		int indexOfFirstElement = *(ia);
		real* au = matrix->upperElements->data + *(ia);
		int nextLineIndex = *(ia + 1);
		int j = i - (nextLineIndex - indexOfFirstElement);

		for (int k = indexOfFirstElement; k < nextLineIndex; k++, j++, au++)
		{
			vector->data[j] -= *(au) * vector->data[i];
		}
	}
	//
}
```

```
void SolveByLowUpDecomposition_DenseMatrix(struct Matrix* matrix, struct
VectorReal* vector)
{
	int i;

	for (i = 0; i < vector->size; i++)
	{
		real_sum sum = 0.0;
		for (int j = 0; j < i; j++)
		{
			sum += matrix->data[i][j] * vector->data[j];
		}
		vector->data[i] = (vector->data[i] - sum) / matrix->data[i][i];
	}

	for (i -= 1; i >= 0; i--)
	{
		real_sum sum = 0.0;
		for (int j = i + 1; j < vector->size; j++)
		{
			sum += matrix->data[i][j] * vector->data[j];
		}
		vector->data[i] -= sum;
	}
}

void SolveByGauss(struct Matrix* matrix, struct VectorReal* vector)
{
	int i;

	for (i = 0; i < matrix->size; i++)// идем по столбцам
	{
		//Find line of the main element
		real mainElement = 0.0;
		int lineOfMainElement = 0;

		for (int j = i; j < matrix->size; j++)
		{
			if (mainElement < fabs(matrix->data[j][i]))
			{
				mainElement = matrix->data[j][i];
				lineOfMainElement = j;
			}
		}
		//

		//Swap lines
		if (lineOfMainElement != i)
		{
			real temp = vector->data[i];
			vector->data[i] = vector->data[lineOfMainElement];
			vector->data[lineOfMainElement] = temp;

			for (int j = i; j < matrix->size; j++)
			{
				temp = matrix->data[i][j];
				matrix->data[i][j] =
matrix->data[lineOfMainElement][j];
				matrix->data[lineOfMainElement][j] = temp;
			}
		}
		//
```

```
            //Exclude elements from column
            for (int j = i + 1; j < matrix->size; j++)
            {
                  //mainElement = matrix->data[j][i];
                  real test = matrix->data[j][i] / mainElement;

                  for (int k = i; k < matrix->size; k++)
                  {
                        matrix->data[j][k] -= test * matrix->data[i][k];
                  }

                  vector->data[j] -= test * vector->data[i];
            }
            //

            //Normalize
            vector->data[i] /= mainElement;

            for (int j = i + 1; j < matrix->size; j++)
            {
                  matrix->data[i][j] /= mainElement;
            }
            //


      }

      //System solution
      for (i -= 2; i >= 0; i--)
      {
            for (int j = i + 1; j < matrix->size; j++)
            {
                  vector->data[i] -= vector->data[j] * matrix->data[i][j];
            }
      }
      //
}
```

Test.h

```
#include "common.h"
#include "io.h"
#include "LinearAlgebra.h"
#include "LowUpDecomposition.h"
#include "SystemOfEquation.h"

void ProfileMatrix_LowUpDecomposition_Test();

void HilbertMatrix_LowUpDecomposition_Test();

void DenseMatrix_Gauss_Test();
```

Test.c

```
#include "Tests.h"
#include <stddef.h>
#include <stdio.h>

void ProfileMatrix_LowUpDecomposition_Test()
{
      struct ProfileMatrix* matrix = NULL;

      struct VectorReal* vector = NULL;
```

```
	matrix = CreateProfileMatrix();
	vector = CreateVectorReal();

	ReadVectorReal(matrix->diagonal, "Test\\di.txt");
	ReadVectorReal(matrix->lowerElements, "Test\\al.txt");
	ReadVectorReal(matrix->upperElements, "Test\\au.txt");
	ReadVectorInt(matrix->indexArray, "Test\\ia.txt");
	ReadVectorReal(vector, "Test\\vector.txt");

	MatrixVectorMultiply_ProfileMatrix(matrix, vector);

	WriteVectorReal(vector, "Test\\right.txt");

	DecomposeProfileMatrix(matrix);

	SolveByLowUpDecomposition_ProfileMatrix(matrix, vector);

	WriteVectorReal(vector, "Test\\profile results.txt");

	ClearProfileMatrix(matrix);
	ClearVectorReal(vector);
}

void HilbertMatrix_LowUpDecomposition_Test()
{
	const int K_MAX = 15;
	struct Matrix* matrix = NULL;
	struct VectorReal* vector = NULL;

	for (int k = 0; k < K_MAX; k++)
	{
		matrix = CreateDenseMatrix();
		vector = CreateVectorReal();

		GenerateHilbertMatrix(matrix, k);
		GenerateVector(vector, k);
		MatrixVectorMultiply_DenseMatrix(matrix, vector);
		WriteVectorReal(vector, "Test\\right.txt");

		DecomposeDenseMatrix(matrix);

		SolveByLowUpDecomposition_DenseMatrix(matrix, vector);
		WriteVectorReal(vector, "Test\\hilbert results.txt");

		ClearDenseMatrix(matrix);
		ClearVectorReal(vector);
	}
}

void DenseMatrix_Gauss_Test()
{
	struct Matrix* matrix = NULL;
	struct VectorReal* vector = NULL;

	matrix = CreateDenseMatrix();
	vector = CreateVectorReal();

	ReadDenseMatrix(matrix, "Test\\dense.txt");
	ReadVectorReal(vector, "Test\\vector.txt");

	MatrixVectorMultiply_DenseMatrix(matrix, vector);

	WriteVectorReal(vector, "Test\\right.txt");
```

```
	SolveByGauss(matrix, vector);

	WriteVectorReal(vector, "Test\\gauss results.txt");

	ClearDenseMatrix(matrix);
	ClearVectorReal(vector);
}
```

Оценка влияния числа обусловленности на точность решения

Исходная матрица:

6 + 10^(-k)	-2	0	0	-4	0	0	0	0	0		1		-17 + 10^(-k)
0	4	0	0	-4	0	0	0	0	0		2		-12
0	0	4	-4	0	0	0	0	0	0		3		-4
0	-1	-1	7	-1	-3	-1	0	0	0		4		-7
0	0	-2	-2	8	-4	0	0	0	0	*	5	=	2
0	0	-4	0	0	4	0	0	0	0		6		12
-1	0	0	-2	-4	-1	14	-3	-1	-2		7		10
0	0	0	0	0	0	-1	2	0	-1		8		-1
0	0	0	0	0	-4	-2	-1	7	0		9		17
0	0	0	-1	-4	0	0	0	0	5		10		26

k	x^k (одинарная точность)	$x^* - x^k$ (одинарная точность)	x^k (двойная точность)	$x^* - x^k$ (двойная точность)	x^k (скалярное произведение)	$x^* - x^k$ (скалярное произведение)
0	0.9999503 1.9999418 2.9999418 3.9999418 4.9999418 5.9999413 6.9999418 7.9999409 8.9999418 9.9999409	4.97e-05 5.82e-05 5.82e-05 5.82e-05 5.82e-05 5.87e-05 5.82e-05 5.91e-05 5.82e-05 5.91e-05	0.9999999999999210 1.9999999999999076 2.9999999999999085 3.9999999999999085 4.9999999999999076 5.9999999999999085 6.9999999999999094 7.9999999999999094 8.9999999999999076 9.9999999999999076	7.90e-14 9.24e-14 9.15e-14 9.15e-14 9.24e-14 9.15e-14 9.06e-14 9.06e-14 9.24e-14 9.24e-14	1.0000640 2.0000744 3.0000744 4.0000744 5.0000744 6.0000739 7.0000734 8.0000744 9.0000734 10.0000744	6.40e-05 7.44e-05 7.44e-05 7.44e-05 7.44e-05 7.39e-05 7.34e-05 7.44e-05 7.34e-05 7.44e-05
1	0.9996715 1.9996662 2.9996662 3.9996662 4.9996662 5.9996662 6.9996667 7.9996657 8.9996662 9.9996662	3.28e-04 3.34e-04 3.34e-04 3.34e-04 3.34e-04 3.34e-04 3.33e-04 3.34e-04 3.34e-04 3.34e-04	0.9999999999985466 1.9999999999985221 2.9999999999985230 3.9999999999985230 4.9999999999985221 5.9999999999985230 6.9999999999985238 7.9999999999985238 8.9999999999985221 9.9999999999985221	1.45e-12 1.48e-12 1.48e-12 1.48e-12 1.48e-12 1.48e-12 1.48e-12 1.48e-12 1.48e-12 1.48e-12	1.0005291 2.0005379 3.0005379 4.0005379 5.0005379 6.0005379 7.0005374 8.0005379 9.0005379 10.0005379	5.29e-04 5.38e-04 5.38e-04 5.38e-04 5.38e-04 5.38e-04 5.37e-04 5.38e-04 5.38e-04 5.38e-04

2	0.9959074 1.9959006 2.9959006 3.9959006 4.9959006 5.9959006 6.9959011 7.9959011 8.9959002 9.9959002	4.09e-03 4.10e-03 4.10e-03 4.10e-03 4.10e-03 4.10e-03 4.10e-03 4.10e-03 4.10e-03 4.10e-03	0.9999999999855318 1.9999999999855076 2.9999999999855076 3.9999999999855076 4.9999999999855076 5.9999999999855076 6.9999999999855085 7.9999999999855067 8.9999999999855085 9.9999999999855049	1.45e-11 1.45e-11 1.45e-11 1.45e-11 1.45e-11 1.45e-11 1.45e-11 1.45e-11 1.45e-11 1.45e-11	1.0056287 2.0056381 3.0056376 4.0056376 5.0056381 6.0056376 7.0056372 8.0056381 9.0056372 10.0056381	5.63e-03 5.64e-03 5.64e-03 5.64e-03 5.64e-03 5.64e-03 5.64e-03 5.64e-03 5.64e-03 5.64e-03
3	0.9504213 1.9504132 2.9504132 3.9504132 4.9504132 5.9504132 6.9504137 7.9504137 8.9504137 9.9504128	4.96e-02 4.96e-02 4.96e-02 4.96e-02 4.96e-02 4.96e-02 4.96e-02 4.96e-02 4.96e-02 4.96e-02	0.9999999998477340 1.9999999998477085 2.9999999998477085 3.9999999998477085 4.9999999998477085 5.9999999998477085 6.9999999998477094 7.9999999998477085 8.9999999998477112 9.9999999998477076	1.52e-10 1.52e-10 1.52e-10 1.52e-10 1.52e-10 1.52e-10 1.52e-10 1.52e-10 1.52e-10 1.52e-10	1.0490866 2.0490947 3.0490947 4.0490947 5.0490947 6.0490947 7.0490942 8.0490952 9.0490942 10.0490952	4.91e-02 4.91e-02 4.91e-02 4.91e-02 4.91e-02 4.91e-02 4.91e-02 4.91e-02 4.91e-02 4.91e-02
4	0.9999987 1.9999990 2.9999990 3.9999990 4.9999990 5.9999986 6.9999990 8.0000000 8.9999990 10.0000000	1.31e-06 9.54e-07 9.54e-07 9.54e-07 9.54e-07 1.43e-06 9.54e-07 0.00e+00 9.54e-07 0.00e+00	0.9999999986296612 1.9999999986296384 2.9999999986296384 3.9999999986296384 4.9999999986296384 5.9999999986296384 6.9999999986296393 7.9999999986296393 8.9999999986296384 9.9999999986296384	1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09	1.3820164 2.3820229 3.3820229 4.3820229 5.3820229 6.3820229 7.3820224 8.3820229 9.3820229 10.3820229	3.82e-01 3.82e-01 3.82e-01 3.82e-01 3.82e-01 3.82e-01 3.82e-01 3.82e-01 3.82e-01 3.82e-01
5	2.9999967 4.0000000 5.0000000 6.0000000 7.0000000 7.9999995 9.0000000 10.0000000 11.0000000 12.0000000	2.00e+00 2.00e+00 2.00e+00 2.00e+00 2.00e+00 2.00e+00 2.00e+00 2.00e+00 2.00e+00 2.00e+00	0.9999999893418539 1.9999999893418359 2.9999999893418359 3.9999999893418359 4.9999999893418359 5.9999999893418359 6.9999999893418368 7.9999999893418368 8.9999999893418359 9.9999999893418359	1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08	-15.4999723 -14.5000000 -13.5000000 -12.5000000 -11.5000000 -10.4999990 -9.4999981 -8.4999990 -7.4999990 -6.5000000	1.65e+01 1.65e+01 1.65e+01 1.65e+01 1.65e+01 1.65e+01 1.65e+01 1.65e+01 1.65e+01 1.65e+01
6			0.9999998781926714 1.9999998781926509 2.9999998781926518 3.9999998781926518 4.9999998781926509 5.9999998781926518 6.9999998781926527 7.9999998781926527 8.9999998781926518 9.9999998781926518	1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07		

7			0.9999985535385324 1.9999985535385081 2.9999985535385081 3.9999985535385081 4.9999985535385081 5.9999985535385081 6.9999985535385090 7.9999985535385072 8.9999985535385072 9.9999985535385072	1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06		
8			0.9999855354307980 1.9999855354307741 2.9999855354307750 3.9999855354307750 4.9999855354307741 5.9999855354307750 6.9999855354307758 7.9999855354307732 8.9999855354307741 9.9999855354307723	1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05		
9			0.9998325225904471 1.9998325225904194 2.9998325225904194 3.9998325225904194 4.9998325225904194 5.9998325225904194 6.9998325225904203 7.9998325225904194 8.9998325225904185 9.9998325225904185	1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04		
10			0.9987823439878446 1.9987823439878243 2.9987823439878243 3.9987823439878243 4.9987823439878243 5.9987823439878243 6.9987823439878252 7.9987823439878234 8.9987823439878234 9.9987823439878234	1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03		
11			0.9871065604854213 1.9871065604854001 2.9871065604854001 3.9871065604854001 4.9871065604854001 5.9871065604854001 6.9871065604854010 7.9871065604853992 8.9871065604854010 9.9871065604853992	1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02		

12			0.8970588235294294 1.8970588235294121 2.8970588235294121 3.8970588235294121 4.8970588235294121 5.8970588235294121 6.8970588235294130 7.8970588235294121 8.8970588235294148 9.8970588235294112	1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01		
13			0.0000000000000165 1.0000000000000000 2.0000000000000009 3.0000000000000009 4.0000000000000000 5.0000000000000009 6.0000000000000018 7.0000000000000009 8.0000000000000018 9.0000000000000000	1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00		

Вывод:

Исходная матрица при $k = \infty$ будет вырожеденной (определитель этой матрицы будет равен нулю). Т.к числа с плавающей точкой представимы конечным образом (число знаков мантиссы ограничено), то существует $k_0 < \infty$ такой, что 10^{-k_0} не будет помещаться в мантиссу числа (т.е просто отбросится), при этом матрица получится вырожденной и решение “сломается”. Очевидно, что чем больше размер мантиссы, тем больше будет и k_0, поэтому решение с одинарной точностью быстрее “ломается”, чем решение с двойной точностью.

Исследования на матрицах Гильберта

k	x^k (одинарная точность)	$x^* - x^k$ (одинарная точность)	x^k (двойная точность)	$x^* - x^k$ (двойная точность)	x^k (скалярное произведение)	$x^* - x^k$ (скалярное произведение)
0	1.0000000	0.00e+00	1.0000000000000000	0.00e+00	1.0000000	0.00e+00
1	0.9999996 2.0000007	4.00e-07 7.00e-07	1.0000000000000007 1.9999999999999987	6.66e-16 1.33e-15	0.9999996 2.0000007	4.00e-07 7.00e-07
2	0.9999976 2.0000117 2.9999893	2.40e-06 1.17e-05 1.07e-05	1.0000000000000060 1.9999999999999702 3.0000000000000275	6.00e-15 2.98e-14 2.75e-14	0.9999995 2.0000010 3.0000000	5.00e-07 1.00e-06 0.00e+00
3	1.0000284 1.9996786 3.0007720 3.9994993	2.84e-05 3.21e-04 7.72e-04 5.01e-04	1.0000000000000493 1.9999999999994236 3.0000000000014166 3.9999999999990674	4.93e-14 5.76e-13 1.42e-12 9.33e-13	1.0000222 1.9997470 3.0006099 3.9996037	2.22e-05 2.53e-04 6.10e-04 3.96e-04
4	1.0001144 1.9976196 3.0108566 3.9830160 5.0085135	1.14e-04 2.38e-03 1.09e-02 1.70e-02 8.51e-03	0.9999999999998956 2.0000000000019513 2.9999999999916991 4.0000000000123563 4.9999999999940332	1.04e-13 1.95e-12 8.30e-12 1.24e-11 5.97e-12	0.9998720 2.0022860 2.9903951 4.0142593 4.9931087	1.28e-04 2.29e-03 9.60e-03 1.43e-02 6.89e-03
5	0.9976134 2.0696220 2.5215759 5.2587738 3.5983696 6.5560975	2.39e-03 6.96e-02 4.78e-01 1.26e+00 1.40e+00 5.56e-01	1.0000000000003868 1.9999999999892388 3.0000000000731486 3.9999999998077094 5.0000000002146106 5.9999999999145697	3.87e-13 1.08e-11 7.31e-11 1.92e-10 2.15e-10 8.54e-11	0.9982065 2.0513437 2.6514726 4.9090810 3.9943027 6.3969998	1.79e-03 5.13e-02 3.49e-01 9.09e-01 1.01e+00 3.97e-01
6	0.9858217 2.5294323 -1.8510780 22.1256180 -27.1714287 33.0598221 -1.6842105	1.42e-02 5.29e-01 4.85e+00 1.81e+01 3.22e+01 2.71e+01 8.68e+00	0.9999999999886949 2.0000000004461036 2.9999999957538233 4.0000000163321090 4.9999999703313520 6.0000000254374619 6.9999999917033966	1.13e-11 4.46e-10 4.25e-09 1.63e-08 2.97e-08 2.54e-08 8.30e-09	0.9909106 2.3461790 -0.2267222 16.2354832 -16.9956169 24.7071915 0.9377273	9.09e-03 3.46e-01 3.23e+00 1.22e+01 2.20e+01 1.87e+01 6.06e+00
7			1.0000000000751492 1.9999999958601160 3.0000000549790542 3.9999996990683329 5.0000008167746799 5.9999988373032451 7.0000008312808681 7.9999997645984466	7.51e-11 4.14e-09 5.50e-08 3.01e-07 8.17e-07 1.16e-06 8.31e-07 2.35e-07		

8			1.0000000003600271 1.9999999759907041 3.0000003953739469 3.9999972403153876 5.0000099339034847 5.9999800374595651 7.0000226141624466 7.9999865022705094 9.0000033003484621	3.60e-10 2.40e-08 3.95e-07 2.76e-06 9.93e-06 2.00e-05 2.26e-05 1.35e-05 3.30e-06		
9			1.0000000055025140 1.9999995212095536 3.0000102435747955 3.9999065996121672 5.0004464050331876 5.9987710994965084 7.0020182642565487 7.9980481971011130 9.0010251968156894 9.9997744634852186	5.50e-09 4.79e-07 1.02e-05 9.34e-05 4.46e-04 1.23e-03 2.02e-03 1.95e-03 1.03e-03 2.26e-04		
10			1.0000000337074626 1.9999964819396752 3.0000909342943949 3.9989871333924825 5.0060120047767072 5.9789361734160025 7.0457131582693933 7.9378698725880383 9.0514620717158607 9.9762526032351033 11.0046795518548581	3.37e-08 3.52e-06 9.09e-05 1.01e-03 6.01e-03 2.11e-02 4.57e-02 6.21e-02 5.15e-02 2.37e-02 4.68e-03		
11			1.0000001614284348 1.9999797098905026 3.0006334792040796 3.9914234724183402 5.0625244093177031 5.7266266431498707 7.7583776759320813 6.6325710155318802 10.5975398882787886 8.8336858694641478 11.4835298270528483 11.9131077498144560	1.61e-07 2.03e-05 6.33e-04 8.58e-03 6.25e-02 2.73e-01 7.58e-01 1.37e+00 1.60e+00 1.17e+00 4.84e-01 8.69e-02		
12			0.9999996096812334 2.0000632619168748 2.9975064672342100 4.0421876311449125 4.6165711083230718 8.0993732997587813 -0.3790755343449064 25.2147653065961777 -17.9437484672267829 37.9685418603236826	3.90e-07 6.33e-05 2.49e-03 4.22e-02 3.83e-01 2.10e+00 7.38e+00 1.72e+01 2.69e+01 2.80e+01 1.85e+01		

			-7.4673215863480777 19.0210716213514672 11.8300650103774121	7.02e+00 1.17e+00		
13			0.9999988063331632 2.0001498806742566 2.9955937825985117 4.0499181640219390 4.7816833125191351 5.7461258214660802 13.9747570535822092 -26.2450521222882003 100.6209100602881108 -142.9221537065374150 174.6227637198911680 -97.3925132066265462 54.6913998512359925 7.0764170819851682	1.19e-06 1.50e-04 4.41e-03 4.99e-02 2.18e-01 2.54e-01 6.97e+00 3.42e+01 9.16e+01 1.53e+02 1.64e+02 1.09e+02 4.17e+01 6.92e+00		

Вывод:

По мере увеличения размерности матрицы Гильберта, “новые” элементы будут приближаться к нулю, а т.к количество знаков мантиссы ограничено, то точность решения будет быстро падать. Также на точность влияет и невозможность представления бесконечных десятичных дробей (из-за конечного представления чисел).

Сравнение алгоритмов по скорости

В качестве примера возьмем разложение LL^T, предполагая, что этот алгоритм имеет меньшую сложность.

Рассмотрим разложение Холецкого (LL^T):

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

Элементы матрицы L можно вычислить по следующим формулам:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad i \in [1,\ n]$$

$$l_{ji} = \frac{1}{l_{ii}} \left[a_{ji} - \sum_{k=1}^{i-1} l_{ik} l_{jk} \right] \quad i \in [1,\ n],\ j \in [1,\ i - 1]$$

Сложность алгоритма:

Метод	L	U	Прямой ход	Обратный ход	Итог
LU	$\frac{n^3}{3} + O(n^2)$	$\frac{n^3}{3} + O(n^2)$	$O(n^2)$	$O(n^2)$	$\frac{2n^3}{3} + O(n^2)$
LL^T	$\frac{n^3}{3} + O(n^2)$	-	$O(n^2)$	$O(n^2)$	$\frac{n^3}{3} + O(n^2)$

В итоге решение СЛАУ разложением Холецкого примерно в 2 раза быстрее, чем разложением через LU. Поэтому, если заранее известно, что разложение Холецкого существует, т.е исходная матрица является симметричной и положительно-определенной матрицы, то целесообразно будет использовать именно этот алгоритм.

Сравнение с методом Гаусса

Будем исследовать именно метод Гаусса с ведущим элементом.

Сравнение по точности решения (двойная точность):

k	Метод Гаусса		LU	
0	1.0000000000000231	2.31e-14	0.9999999999999210	7.90e-14
	2.0000000000000266	2.66e-14	1.9999999999999076	9.24e-14
	3.0000000000000266	2.66e-14	2.9999999999999085	9.15e-14
	4.0000000000000266	2.66e-14	3.9999999999999085	9.15e-14
	5.0000000000000266	2.66e-14	4.9999999999999076	9.24e-14
	6.0000000000000266	2.66e-14	5.9999999999999085	9.15e-14
	7.0000000000000275	2.75e-14	6.9999999999999094	9.06e-14
	8.0000000000000266	2.66e-14	7.9999999999999094	9.06e-14
	9.0000000000000284	2.84e-14	8.9999999999999076	9.24e-14
	10.0000000000000266	2.66e-14	9.9999999999999076	9.24e-14
1	0.9999999999997975	2.03e-13	0.9999999999985466	1.45e-12
	1.9999999999997939	2.06e-13	1.9999999999985221	1.48e-12
	2.9999999999997939	2.06e-13	2.9999999999985230	1.48e-12
	3.9999999999997939	2.06e-13	3.9999999999985230	1.48e-12
	4.9999999999997939	2.06e-13	4.9999999999985221	1.48e-12
	5.9999999999997939	2.06e-13	5.9999999999985230	1.48e-12
	6.9999999999997948	2.05e-13	6.9999999999985238	1.48e-12
	7.9999999999997948	2.05e-13	7.9999999999985238	1.48e-12
	8.9999999999997957	2.04e-13	8.9999999999985221	1.48e-12
	9.9999999999997939	2.06e-13	9.9999999999985221	1.48e-12
2	0.9999999999997740	2.26e-13	0.9999999999855318	1.45e-11
	1.9999999999997735	2.26e-13	1.9999999999855076	1.45e-11
	2.9999999999997740	2.26e-13	2.9999999999855076	1.45e-11
	3.9999999999997740	2.26e-13	3.9999999999855076	1.45e-11
	4.9999999999997735	2.26e-13	4.9999999999855076	1.45e-11
	5.9999999999997735	2.26e-13	5.9999999999855076	1.45e-11
	6.9999999999997753	2.25e-13	6.9999999999855085	1.45e-11
	7.9999999999997735	2.26e-13	7.9999999999855067	1.45e-11
	8.9999999999997744	2.26e-13	8.9999999999855085	1.45e-11
	9.9999999999997744	2.26e-13	9.9999999999855049	1.45e-11
3	0.9999999999707336	2.93e-11	0.9999999998477340	1.52e-10
	1.9999999999707292	2.93e-11	1.9999999998477085	1.52e-10
	2.9999999999707292	2.93e-11	2.9999999998477085	1.52e-10
	3.9999999999707292	2.93e-11	3.9999999998477085	1.52e-10
	4.9999999999707292	2.93e-11	4.9999999998477085	1.52e-10
	5.9999999999707292	2.93e-11	5.9999999998477085	1.52e-10
	6.9999999999707310	2.93e-11	6.9999999998477094	1.52e-10
	7.9999999999707301	2.93e-11	7.9999999998477085	1.52e-10
	8.9999999999707310	2.93e-11	8.9999999998477112	1.52e-10
	9.9999999999707292	2.93e-11	9.9999999998477076	1.52e-10

4	0.9999999992345998 1.9999999992345874 2.9999999992345874 3.9999999992345874 4.9999999992345874 5.9999999992345874 6.9999999992345892 7.9999999992345865 8.9999999992345874 9.9999999992345874	7.65e-10 7.65e-10 7.65e-10 7.65e-10 7.65e-10 7.65e-10 7.65e-10 7.65e-10 7.65e-10 7.65e-10	0.9999999986296612 1.9999999986296384 2.9999999986296384 3.9999999986296384 4.9999999986296384 5.9999999986296384 6.9999999986296393 7.9999999986296393 8.9999999986296384 9.9999999986296384	1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09 1.37e-09
5	0.9999999986492933 1.9999999986492911 2.9999999986492920 3.9999999986492920 4.9999999986492911 5.9999999986492911 6.9999999986492947 7.9999999986492929 8.9999999986492938 9.9999999986492920	1.35e-09 1.35e-09 1.35e-09 1.35e-09 1.35e-09 1.35e-09 1.35e-09 1.35e-09 1.35e-09 1.35e-09	0.9999999893418539 1.9999999893418359 2.9999999893418359 3.9999999893418359 4.9999999893418359 5.9999999893418359 6.9999999893418368 7.9999999893418368 8.9999999893418359 9.9999999893418359	1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08 1.07e-08
6	0.9999999133297472 1.9999999133297326 2.9999999133297326 3.9999999133297326 4.9999999133297326 5.9999999133297317 6.9999999133297361 7.9999999133297344 8.9999999133297326 9.9999999133297326	8.67e-08 8.67e-08 8.67e-08 8.67e-08 8.67e-08 8.67e-08 8.67e-08 8.67e-08 8.67e-08 8.67e-08	0.9999998781926714 1.9999998781926509 2.9999998781926518 3.9999998781926518 4.9999998781926509 5.9999998781926518 6.9999998781926527 7.9999998781926527 8.9999998781926518 9.9999998781926518	1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07 1.22e-07
7	0.9999991558093018 1.9999991558092871 2.9999991558092871 3.9999991558092871 4.9999991558092871 5.9999991558092871 6.9999991558092880 7.9999991558092871 8.9999991558092862 9.9999991558092862	8.44e-07 8.44e-07 8.44e-07 8.44e-07 8.44e-07 8.44e-07 8.44e-07 8.44e-07 8.44e-07 8.44e-07	0.9999985535385324 1.9999985535385081 2.9999985535385081 3.9999985535385081 4.9999985535385081 5.9999985535385081 6.9999985535385090 7.9999985535385072 8.9999985535385072 9.9999985535385072	1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06 1.45e-06
8	0.9999936967168668 1.9999936967168566 2.9999936967168566 3.9999936967168566 4.9999936967168566 5.9999936967168566 6.9999936967168583 7.9999936967168583 8.9999936967168566 9.9999936967168566	6.30e-06 6.30e-06 6.30e-06 6.30e-06 6.30e-06 6.30e-06 6.30e-06 6.30e-06 6.30e-06 6.30e-06	0.9999855354307980 1.9999855354307741 2.9999855354307750 3.9999855354307750 4.9999855354307741 5.9999855354307750 6.9999855354307758 7.9999855354307732 8.9999855354307741 9.9999855354307723	1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05 1.45e-05

9	0.9999954976756764 1.9999954976756760 2.9999954976756760 3.9999954976756760 4.9999954976756760 5.9999954976756760 6.9999954976756769 7.9999954976756760 8.9999954976756769 9.9999954976756751	4.50e-06 4.50e-06 4.50e-06 4.50e-06 4.50e-06 4.50e-06 4.50e-06 4.50e-06 4.50e-06 4.50e-06	0.9998325225904471 1.9998325225904194 2.9998325225904194 3.9998325225904194 4.9998325225904194 5.9998325225904194 6.9998325225904203 7.9998325225904194 8.9998325225904185 9.9998325225904185	1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04 1.67e-04
10	1.0002701455409051 2.0002701455409095 3.0002701455409095 4.0002701455409095 5.0002701455409095 6.0002701455409104 7.0002701455409104 8.0002701455409095 9.0002701455409078 10.0002701455409095	2.70e-04 2.70e-04 2.70e-04 2.70e-04 2.70e-04 2.70e-04 2.70e-04 2.70e-04 2.70e-04 2.70e-04	0.9987823439878446 1.9987823439878243 2.9987823439878243 3.9987823439878243 4.9987823439878243 5.9987823439878243 6.9987823439878252 7.9987823439878234 8.9987823439878234 9.9987823439878234	1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03 1.22e-03
11	0.9898887765419784 1.9898887765419619 2.9898887765419619 3.9898887765419619 4.9898887765419619 5.9898887765419619 6.9898887765419637 7.9898887765419637 8.9898887765419637 9.9898887765419619	1.01e-02 1.01e-02 1.01e-02 1.01e-02 1.01e-02 1.01e-02 1.01e-02 1.01e-02 1.01e-02 1.01e-02	0.9871065604854213 1.9871065604854001 2.9871065604854001 3.9871065604854001 4.9871065604854001 5.9871065604854001 6.9871065604854010 7.9871065604853992 8.9871065604854010 9.9871065604853992	1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02 1.29e-02
12	1.1218678815489538 2.1218678815489742 3.1218678815489742 4.1218678815489742 5.1218678815489742 6.1218678815489742 7.1218678815489742 8.1218678815489742 9.1218678815489742 10.1218678815489742	1.22e-01 1.22e-01 1.22e-01 1.22e-01 1.22e-01 1.22e-01 1.22e-01 1.22e-01 1.22e-01 1.22e-01	0.8970588235294294 1.8970588235294121 2.8970588235294121 3.8970588235294121 4.8970588235294121 5.8970588235294121 6.8970588235294130 7.8970588235294121 8.8970588235294148 9.8970588235294112	1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01 1.03e-01
13	0.5656565656565715 1.5656565656565649 2.5656565656565649 3.5656565656565649 4.5656565656565649 5.5656565656565649 6.5656565656565675 7.5656565656565675 8.5656565656565657 9.5656565656565657	4.34e-01 4.34e-01 4.34e-01 4.34e-01 4.34e-01 4.34e-01 4.34e-01 4.34e-01 4.34e-01 4.34e-01	0.0000000000000165 1.0000000000000000 2.0000000000000009 3.0000000000000009 4.0000000000000000 5.0000000000000009 6.0000000000000018 7.0000000000000009 8.0000000000000018 9.0000000000000000	1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00 1.00e+00

Сравнение алгоритмов по скорости

Метод	Приведение матрицы	Прямой ход	Обратный ход	Итог
LU	$\frac{2n^3}{3} + O(n^2)$	$O(n^2)$	$O(n^2)$	$\frac{2n^3}{3} + O(n^2)$
метод Гаусса	$\frac{2n^3}{3} + O(n^2)$	-	$O(n^2)$	$\frac{2n^3}{3} + O(n^2)$

Вывод:

По точности методы почти не отличаются и оба неустойчивы к числу обусловленности исходной матрицы. Главным преимуществом LU можно считать возможность хранения матрицы в форматах, использующих меньше памяти, а также возможность проводить вычисления с различными правыми частями. Решение методом Гаусса с ведущим элементом можно использовать в тех случаях, когда матрица невырождена, но при этом на главной диагонали находятся нули.