МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Национальный исследовательский**

**Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Фундаментальная информатика и информационные

технологии»

Магистерская программа: «Инженерия программного обеспечения»

**ОТЧЕТ**

по лабораторной работе

**Выполнил:** студент группы382006-1м

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.А. Солуянов

**Проверил:**

к.ф.-м. н., доц., доцент каф. МОСТ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_К.А. Баркалов

Нижний Новгород

2021

# Введение

Решение систем линейных алгебраических уравнений (СЛАУ) является достаточно важной вычислительной задачей (примерно 75% всех расчетных математических задач приходится на их решение). С решением СЛАУ связаны такие задачи, как вычисление определителей, обращение матриц, вычисление собственных значений и собственных векторов матриц, интерполирование, аппроксимация по методу наименьших квадратов, решение систем дифференциальных уравнений и многие другие. Современная вычислительная математика располагает большим арсеналом методов решения СЛАУ, а математическое обеспечение ЭВМ – многими пакетами программ и программными системами, позволяющими решать СЛАУ.

Методы решения СЛАУ можно разделить на две группы:

1. Прямые методы позволяют найти точное решение системы (метод Крамера, разложение Холецкого, метод Гаусса, LU – разложение и т.д.)
2. Итерационные – позволяют получить решение в результате последовательных приближений (методы Зейделя и Якоби,

Очень важно знать методы решения СЛАУ и уметь их применять.

В данной работе рассматривается прямой метод решения СЛАУ – метод LU – разложения.

# Постановка задачи

## Описание метода

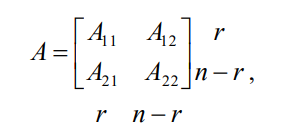
LU-разложение — это представление матрицы в виде , где — нижнетреугольная матрица с единичной диагональю, а — верхнетреугольная матрица. – разложение является модификацией метода Гаусса.

LU-разложение существует только в том случае, когда матрица  {\displaystyle A} обратима, а все ведущие (угловые) главные [миноры](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BD%D0%BE%D1%80_(%D0%BB%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%B0%D1%8F_%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0)) матрицы  {\displaystyle A}[невырождены](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B2%D1%8B%D1%80%D0%BE%D0%B6%D0%B4%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0).

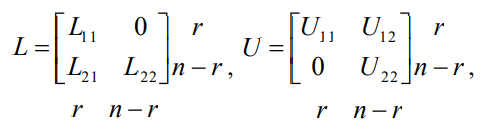
Недостаток стандартного алгоритма LU-разложения обусловлен тем, что его вычисления плохо соответствует правилам использования кэш-памяти − быстродействующей дополнительной памяти компьютера, используемой для хранения копии наиболее часто используемых областей оперативной памяти.

Это связано с тем, что при больших размерах матрицы во время арифметических операций над матрицами зачастую приходится обращаться к элементам, не лежащим вблизи в памяти, что приводит к неэффективному использованию кэша. Возможный способ улучшения ситуации – укрупнение вычислительных операций, приводящее к последовательной обработке некоторых прямоугольных подматриц матрицы

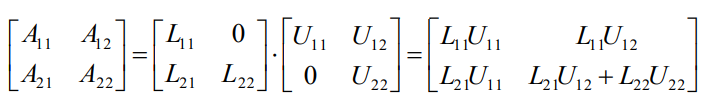
LU-разложение можно организовать так, что матричные операции (реализация которых допускает эффективное использование кэш-памяти) станут основными. Для этого представим матрицу в блочном виде

**

где r – блочный параметр, − подматрица матрицы размера , − размера , − размера , − размера . Компоненты и искомого разложения также запишем в блочном виде



де , , , , , − соответствующего размера подматрицы матриц и . Рассмотрим теперь связь между исходной матрицей и ее разложением в блочном виде.



Блоки и можно найти, применив стандартный метод Гаусса .Затем, решая треугольные системы с несколькими правыми частями будет получено решение для блоков и .

Следующий шаг алгоритма состоит в вычислении редуцированной матрицы , в процессе которого используются ставшие известными блоки и и соотношение .

=

Как следует из данной формулы, LU-разложение редуцированной матрицы совпадает с искомыми блоками матрицы , и для его нахождения можно применить описанный алгоритм рекурсивно.

## Трудоемкость

Приведенная блочная схема требует порядка операций. Оценим долю матричных операций.

Пусть размер матрицы кратен размеру блоку, т.е. .

Операции, не являющиеся матричными, используются при выполнении разложения матрицы на и и требуют операций.

В процессе блочного разложения потребуется решать подобных систем, поэтому доля матричных операций можно оценить как

# Реализация

## Описание алгоритма

Входными параметрами алгоритма является указатель на массив, в котором по строкам хранится матрица и размерность матрицы .

Формат выхода должен выглядеть как два указателя на массивы, в которых по строкам записаны матрицы и .

Весь итерационный процесс можно условно разделить на 4 секции:

1. Подсчет матриц для блоков, располагающихся на диагонали.
2. Вычисление подматрицы , решая верхнюю систему уравнений.
3. Вычисление подматрицы , решая нижнюю систему уравнений.
4. Вычисление редуцированной матрицы

Поэтому удобно выделить эти пункты в отдельные функции (Листинг 1-4).

Листинг 1. Выполнение разложения для диагональных блоков

void DiagonalMatrixDecomposition(int offset, int size, int &N, double\* A, double\* L, double\* U) {

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

for (int j = 0; j < size; j++) {

U[N \* (offset + i) + offset + j] = A[N \* (offset + i) + offset + j];

}

}

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

L[N \* (offset + i) + offset + i] = 1;

for (int k = i + 1; k < size; k++) {

double mu = U[N \* (offset + k) + offset + i] / U[N \* (offset + i) + offset + i];

for (int j = i; j < size; j++) {

U[N \* (offset + k) + offset + j] -= mu \* U[N \* (offset + i) + offset + j];

}

L[N \* (offset + k) + offset + i] = mu;

L[N \* (offset + i) + offset + k] = 0;

}

}

for (int i = 1; i < size; i++) {

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

U[N \* (offset + i) + offset + j] = 0;

}

}

}

Листинг 3. Решение нижней системы уравнений

void SolveLower(int offset, int size, int& N, double\* A, double\* L, double\* U) {

int row\_num;

int col\_num;

#pragma omp parallel for private(row\_num, col\_num)

for (int k = 0; k < N - offset - BLOCK\_SIZE; k++) {

row\_num = (offset + BLOCK\_SIZE + k) \* N;

col\_num = offset;

L[row\_num + col\_num] = A[row\_num + col\_num] / U[offset \* N + col\_num];

for (int i = 1; i < size; i++) {

L[row\_num + col\_num + i] = A[row\_num + col\_num + i];

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

L[row\_num + col\_num + i] -= L[row\_num + j + offset] \* U[(offset + j) \* N + col\_num + i];

}

L[row\_num + col\_num + i] /= U[(offset + i) \* N + col\_num + i];

}

}

}

Листинг 2. Решение верхней системы уравнений

void SolveUpper(int offset, int size, int& N, double\* A, double\* L, double\* U) {

int row\_num;

int col\_num;

#pragma omp parallel for private(row\_num, col\_num)

for (int k = 0; k < N - offset - BLOCK\_SIZE; k++) {

row\_num = offset \* N;

col\_num = offset + BLOCK\_SIZE;

U[row\_num + col\_num + k] = A[row\_num + col\_num + k];

for (int i = 1; i < size; i++) {

U[row\_num + i \* N + col\_num + k] = A[row\_num + i \* N + col\_num + k];

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

U[row\_num + i \* N + col\_num + k] -= L[row\_num + i \* N + j + offset] \* U[row\_num + j \* N + col\_num + k];

}

}

}

}

Листинг 4. Вычисление редуцированной матрицы

void UpdateDiagonalSubmatrix(int offset, int size, int& N, double\* A, double\* L, double\* U) {

int row\_offset;

#pragma omp parallel for if (N - offset > 400) private(row\_offset)

for (int i = 0; i < N - offset - BLOCK\_SIZE; i++) {

row\_offset = (offset + BLOCK\_SIZE + i) \* N;

for (int j = 0; j < N - offset - BLOCK\_SIZE; j++) {

double sum = 0;

for (int k = 0; k < BLOCK\_SIZE; k++) {

sum += L[row\_offset + k + offset] \* U[(offset + k) \* N + offset + BLOCK\_SIZE + j];

}

A[row\_offset + offset + BLOCK\_SIZE + j] -= sum;

}

}

}

В конечном счете целевую функцию можно определить последовательным вызовом всех 4 функций N раз.

## Схема распараллеливания

Так как большая часть алгоритма состоит циклов по обходу матриц, целесообразно применить механизма их распараллеливания.

Для этого была использована библиотека OpenMP, а именно директива «#pragma omp parallel for», с помощью которой циклы были поделены между несколькими потоками для независимого исполнения.

Вычисление верхних и нижних систем линейных уравнений независимо друг от друга, поэтому они могут быть помещены в параллельные секции с помощью директив «#pragma omp parallel sections» и «#pragma omp section»

## Подтверждение корректности алгоритма

# Результаты

## Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (Таблица 1).

|  |  |
| --- | --- |
| Процессор | Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz |
| Память (RAM, Cache) | 8,00 ГБ, L1 – 256 Kb, L2 – 1 Mb, L3 – 8 Mb |
| Операционная система | Windows 10 |
| Среда разработки | Visual Studio 2019 |
| Компилятор | Intel(R) oneAPI DPC++/C++ Compiler 2021.1 |
| Библиотеки | OpenMP |

Таблица 1. Тестовая инфраструктура

## Эксперименты

Заключение

Литература

Приложение