# Министерство образования и науки Российской Федерации

## Федеральное государственное бюджетное образовательное учреждение высшего образования

## «Новосибирский государственный технический университет»

NSTU_Logo_blue

## Кафедра прикладной математики

### Практическое задание № 1 по дисциплине «Численные методы»

**Прямые методы решения СЛАУ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| сигма градиент синий1 | Факультет: | ПМИ |  |  |
| Группа: | ПМ-63 |  |  |
| Студент: | Шепрут И.И. |  |  |
| Вариант: | 11 |  |  |
| Преподаватели: | Задорожный А.Л.  Персова М.Г. |  |  |

Новосибирск

2018

1. **Цель работы**

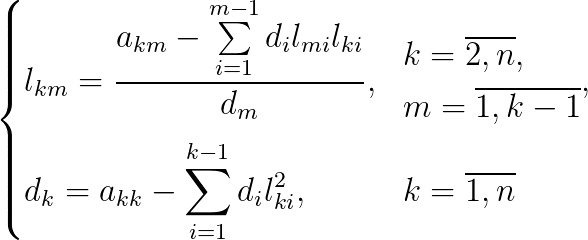
Разработать программу решения СЛАУ прямым методом с хранением матрицы в профильном или ленточном формате. Исследовать накопление погрешности и ее зависимость от числа обусловленности. Сравнить реализованный метод по точности получаемого решения и количеству действий с методом Гаусса.

1. **Анализ задачи**

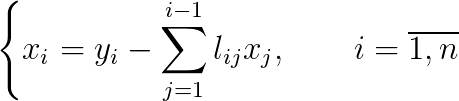
Требования к матрице А: симметричность и невырожденность.

Формулы для решения:

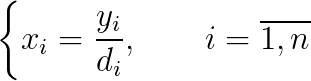
* **Разложение LDLT=A**



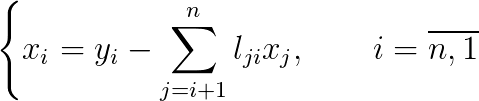
* **Прямой ход**



* **Центральный ход**



* **Обратный ход**



1. **Код программы**

Код программы состоит из файлов:

common.h + common.cpp – пара общих функций и объявление вещественных типов.

matrix.h + matrix.cpp – модуль для работы с матрицами в плотном формате.

sparse.h + sparse.cpp – модуль для работы с матрицами в профильном формате.

numerical\_tests.cpp – программа, которая записывает информацию в файлы для построения следующих 3 таблиц.

matrix.h изначально использовался для понимания работы разложения и т. д., а затем он использовался и для тестов профильного формата.

Для задания текущей точности, необходимо при компиляции указать define для всех файлов:

* ALL\_FLOAT – всё в формате float
* ALL\_DOUBLE – всё в формате double
* ALL\_FLOAT\_WITH\_DOUBLE – всё в формате float, но суммы считаются в double

**common.h:**

|  |
| --- |
| #pragma once  #ifdef ALL\_FLOAT  typedef float real;  typedef float sumreal;  #endif  #ifdef ALL\_DOUBLE  typedef double real;  typedef double sumreal;  #endif  #ifdef ALL\_FLOAT\_WITH\_DOUBLE  typedef float real;  typedef double sumreal;  #endif  #ifndef ALL\_FLOAT  #ifndef ALL\_DOUBLE  #ifndef ALL\_FLOAT\_WITH\_DOUBLE  #error "Type isn't defined"  typedef double real;  typedef double sumreal;  #endif  #endif  #endif  bool isNear(double a, double b);  double random(void);  int intRandom(int min, int max); |

**matrix.h:**

|  |
| --- |
| #pragma once  #include <string>  #include <vector>  #include "common.h"  //-----------------------------------------------------------------------------  class Matrix  {  public:  Matrix(int n = 0, int m = 0, real fill = 0); // n - количество столбцов, m - количество строк  void loadFromFile(std::string fileName);  void saveToFile(std::string fileName) const;  void getFromVector(int n, int m, const std::vector<real>& data);  void resize(int n, int m, real fill = 0);  void negate(void);  bool isSymmetric(void) const;  bool isLowerTriangular(void) const;  bool isUpperTriangular(void) const;  bool isDiagonal(void) const;  bool isDiagonalIdentity(void) const;  bool isDegenerate(void) const;  real& operator()(int i, int j);  const real& operator()(int i, int j) const;  int width(void) const;  int height(void) const;  private:  std::vector<std::vector<real>> m\_matrix;  int m\_n, m\_m;  };  //-----------------------------------------------------------------------------  void generateSparseSymmetricMatrix(  int n,  int min, int max,  real percent,  Matrix& result  );  void generateLMatrix(  int n,  int min, int max,  real percent,  Matrix& result  );  void generateDiagonalMatrix(  int n,  int min, int max,  Matrix& result  );  void generateVector(  int n,  int min, int max,  Matrix& result  );  void generateVector(  int n,  Matrix& result  );  void generateGilbertMatrix(int n, Matrix& result);  void generateTestMatrix(int n, int profileSize, Matrix& result);  //-----------------------------------------------------------------------------  bool mul(const Matrix& a, const Matrix& b, Matrix& result);  bool sum(const Matrix& a, const Matrix& b, Matrix& result);  sumreal sumAllElementsAbs(const Matrix& a);  bool transpose(Matrix& a);  bool calcLDL(const Matrix& a, Matrix& l, Matrix& d);  bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x);  bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x);  bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x);  bool solveSLAE\_by\_LDL(const Matrix& a, const Matrix& y, Matrix& x);  bool solveSLAE\_byGaussMethod(const Matrix& a, const Matrix& y, Matrix& x); |

**sparse.h:**

|  |
| --- |
| #pragma once  #include <string>  #include <vector>  #include "matrix.h"  #include "common.h"  //-----------------------------------------------------------------------------  /\*\* Симметричная матрица в профильном формате. \*/  class MatrixProfileSymmetric  {  public:  MatrixProfileSymmetric();  MatrixProfileSymmetric(const Matrix& a);  void loadFromFile(std::string fileName);  void saveToFile(std::string fileName) const;  void generate(int n, int min, int max, int maxDistanceToDiagonal);  void toDenseMatrix(Matrix& dense) const;  //-------------------------------------------------------------------------  void negate(void);  //-------------------------------------------------------------------------  int size(void) const;  real& getDiagonalElement(int n);  const real& getDiagonalElement(int n) const;  //-------------------------------------------------------------------------  int getLineFirstElementPos(int lineNo) const;  int getLineSize(int lineNo) const;  // Нумерация начинается с 0, но дается элемент с индексом в плотной матрице как getLineFirstElementPos  real& getLineElement(int lineNo, int elemNo);  const real& getLineElement(int lineNo, int elemNo) const;  std::vector<real>::iterator getLineFirstElement(int lineNo);  std::vector<real>::const\_iterator getLineFirstElement(int lineNo) const;  private:  std::vector<real> di;  std::vector<real> al;  std::vector<int> ai;  };  //-----------------------------------------------------------------------------  class Vector  {  public:  Vector();  Vector(int size, real fill = 0);  Vector(const Matrix& a);  void loadFromFile(std::string fileName);  void saveToFile(std::string fileName) const;  void generate(int n, int min, int max);  void generate(int n);  //-------------------------------------------------------------------------  void resize(int n, real fill = 0);  void negate(void);  void toDenseMatrix(Matrix& dense, bool isVertical) const;  //-------------------------------------------------------------------------  int size(void) const;    real& operator()(int i);  const real& operator()(int i) const;  private:  std::vector<real> mas;  };  //-----------------------------------------------------------------------------  bool mul(const MatrixProfileSymmetric& a, const Vector& x, Vector& y);  bool sum(const Vector& a, const Vector& b, Vector& result);  sumreal sumAllElementsAbs(const Vector& a);  void calcLDL(MatrixProfileSymmetric& a\_l);  void calcGaussianReverseOrder(const MatrixProfileSymmetric& l, Vector& y\_x);  void calcGaussianFrontOrder(const MatrixProfileSymmetric& l, Vector& y\_x);  void calcGaussianCentralOrder(const MatrixProfileSymmetric& d, Vector& y\_x);  void solveSLAE\_by\_LDL(MatrixProfileSymmetric& a, Vector& y\_x); |

**common.cpp:**

|  |
| --- |
| #include <cmath>  #include "common.h"  //-----------------------------------------------------------------------------  bool isNear(double a, double b) {  if (a != 0) {  if (fabs(a - b)/a > 0.0001)  return false;  } else {  if (fabs(b) > 0.0001)  return false;  }  return true;  }  //-----------------------------------------------------------------------------  double random(void) {  return std::rand() / double(RAND\_MAX);  }  //-----------------------------------------------------------------------------  int intRandom(int min, int max) {  return min + random() \* (max - min);  } |

**matrix.cpp:**

|  |
| --- |
| #include <fstream>  #include <iomanip>  #include "matrix.h"  //-----------------------------------------------------------------------------  Matrix::Matrix(int n, int m, real fill) : m\_matrix(m, std::vector<real>(n, fill)), m\_n(n), m\_m(m) {  }  //-----------------------------------------------------------------------------  void Matrix::loadFromFile(std::string fileName) {  std::ifstream fin(fileName);  m\_matrix.clear();  int n, m;  fin >> n >> m;  resize(n, m);  for (int i = 0; i < height(); ++i) {  for (int j = 0; j < width(); ++j) {  fin >> operator()(i, j);  }  }  fin.close();  }  //-----------------------------------------------------------------------------  void Matrix::saveToFile(std::string fileName) const {  std::ofstream fout(fileName);  fout << m\_n << '\t' << m\_m << std::endl;  fout.precision(std::numeric\_limits<real>::digits10);  int w = std::numeric\_limits<real>::digits10 + 4;  for (int i = 0; i < height(); ++i) {  for (int j = 0; j < width(); ++j)  fout << std::setw(w) << operator()(i, j);  fout << std::endl;  }  fout.close();  }  //-----------------------------------------------------------------------------  void Matrix::getFromVector(int n, int m, const std::vector<real>& data) {  resize(n, m);  for (int i = 0; i < data.size(); ++i)  operator()(i / m, i % n) = data[i];  }  //-----------------------------------------------------------------------------  void Matrix::resize(int n, int m, real fill) {  if (m\_n != n || m\_m != m) {  m\_n = n;  m\_m = m;  m\_matrix.clear();  m\_matrix.resize(m\_m, std::vector<real>(m\_n, fill));  }  }  //-----------------------------------------------------------------------------  void Matrix::negate(void) {  for (auto& i : m\_matrix) {  for (auto& j : i) {  j = -j;  }  }  }  //-----------------------------------------------------------------------------  bool Matrix::isSymmetric(void) const {  if (height() != width())  return false;  for (int i = 0; i < height(); ++i) {  for (int j = 0; j <= i; ++j) {  const real& a = operator()(i, j);  const real& b = operator()(j, i);  if (!isNear(a, b))  return false;  }  }  return true;  }  //-----------------------------------------------------------------------------  bool Matrix::isLowerTriangular(void) const {  if (height() != width())  return false;  for (int i = 0; i < height(); ++i) {  for (int j = 0; j < i; ++j) {  if (fabs(operator()(j, i)) > 0.000001)  return false;  }  }    return true;  }  //-----------------------------------------------------------------------------  bool Matrix::isUpperTriangular(void) const {  if (height() != width())  return false;  for (int i = 0; i < height(); ++i) {  for (int j = 0; j < i; ++j) {  if (operator()(i, j) != 0)  return false;  }  }    return true;  }  //-----------------------------------------------------------------------------  bool Matrix::isDiagonal(void) const {  if (height() != width())  return false;  for (int i = 0; i < height(); ++i) {  for (int j = 0; j < i; ++j) {  if (operator()(j, i) != 0 && operator()(i, j) != 0)  return false;  }  }    return true;  }  //-----------------------------------------------------------------------------  bool Matrix::isDiagonalIdentity(void) const {  if (height() != width())  return false;  for (int i = 0; i < height(); ++i) {  if (operator()(i, i) != 1)  return false;  }    return true;  }  //-----------------------------------------------------------------------------  bool Matrix::isDegenerate(void) const {  // TODO  return false;  }  //-----------------------------------------------------------------------------  real& Matrix::operator()(int i, int j) {  return m\_matrix[i][j];  }  //-----------------------------------------------------------------------------  const real& Matrix::operator()(int i, int j) const {  return m\_matrix[i][j];  }  //-----------------------------------------------------------------------------  int Matrix::width(void) const {  return m\_n;  }  //-----------------------------------------------------------------------------  int Matrix::height(void) const {  return m\_m;  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  void generateSparseSymmetricMatrix(int n, int min, int max, real percent, Matrix& result) {  result.resize(n, n, 0);  int count = percent \* n \* n;  for (int k = 0; k < count; ++k) {  int i = intRandom(0, n-1);  int j = intRandom(0, i-1);  result(i, j) = intRandom(min, max);  result(j, i) = result(i, j);  }  for (int i = 0; i < n; i++)  result(i, i) = intRandom(1, max - min);  }  //-----------------------------------------------------------------------------  void generateLMatrix(int n, int min, int max, real percent, Matrix& result) {  result.resize(n, n, 0);  int count = percent \* n \* n / 2;  for (int k = 0; k < count; ++k) {  int i = intRandom(0, n-1);  int j = intRandom(0, i-1);  result(i, j) = intRandom(min, max);  }  for (int i = 0; i < n; ++i) {  result(i, i) = 1;  }  }  //-----------------------------------------------------------------------------  void generateDiagonalMatrix(int n, int min, int max, Matrix& result) {  result.resize(n, n, 0);  for (int i = 0; i < n; ++i)  result(i, i) = intRandom(min, max);  }  //-----------------------------------------------------------------------------  void generateVector(int n, int min, int max, Matrix& result) {  result.resize(1, n, 0);  for (int i = 0; i < n; ++i)  result(i, 0) = intRandom(min, max);  }  //-----------------------------------------------------------------------------  void generateVector(int n, Matrix& result) {  result.resize(1, n, 0);  for (int i = 0; i < n; ++i)  result(i, 0) = i+1;  }  //-----------------------------------------------------------------------------  void generateGilbertMatrix(int n, Matrix& result) {  result.resize(n, n);  for (int i = 0; i < n; ++i) {  for (int j = 0; j < n; ++j) {  result(i, j) = double(1.0)/double((i+1)+(j+1)-1);  }  }  }  //-----------------------------------------------------------------------------  void generateTestMatrix(int n, int profileSize, Matrix& result) {  result.resize(n, n);  for (int i = 0; i < n; ++i) {  for (int j = 0; j < profileSize; ++j) if (i-j-1 >= 0) {  result(i, i-j-1) = -intRandom(0, 5);  result(i-j-1, i) = result(i, i-j-1);  }  }  for (int i = 0; i < n; ++i) {  sumreal sum = 0;  for (int j = 0; j < n; ++j) if (i != j) {  sum += result(i, j);  }  result(i, i) = -sum;  }  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  bool mul(const Matrix& a, const Matrix& b, Matrix& result) {  // result = rus\_a \* b  if (a.width() != b.height())  return false;  result.resize(b.width(), a.height());  for (int i = 0; i < b.width(); ++i) {  for (int j = 0; j < a.height(); ++j) {  real sum = 0;  for (int k = 0; k < a.width(); ++k) {  sum += a(j, k) \* b(k, i);  }  result(j, i) = sum;  }  }  return true;  }  //-----------------------------------------------------------------------------  bool sum(const Matrix& a, const Matrix& b, Matrix& result) {  // result = rus\_a + b  if (a.width() != b.width() || a.height() != b.height())  return false;  result.resize(a.width(), a.height());  for (int i = 0; i < a.width(); ++i) {  for (int j = 0; j < a.height(); ++j) {  result(j, i) = a(j, i) + b(j, i);  }  }  return true;  }  //-----------------------------------------------------------------------------  bool transpose(Matrix& a) {  // rus\_a = rus\_a^T  if (a.height() != a.width())  return false;  for (int i = 0; i < a.height(); ++i) {  for (int j = 0; j < i; ++j) {  std::swap(a(j, i), a(i, j));  }  }  return true;  }  //-----------------------------------------------------------------------------  sumreal sumAllElementsAbs(const Matrix& a) {  sumreal sum = 0;  for (int i = 0; i < a.height(); ++i) {  for (int j = 0; j < a.width(); ++j) {  sum += fabs(a(i, j));  }  }  return sum;  }  //-----------------------------------------------------------------------------  bool calcLDL(const Matrix& a, Matrix& l, Matrix& d) {  // l \* d \* l^T = rus\_a  if (!a.isSymmetric())  return false;  l.resize(a.width(), a.height(), 0);  d.resize(a.width(), a.height(), 0);  for (int i = 0; i < a.height(); ++i) {  // Считаем элементы матрицы L  for (int j = 0; j < i; ++j) {  real sum = 0;  for (int k = 0; k < j; ++k)  sum += d(k, k) \* l(j, k) \* l(i, k);  if (fabs(d(j, j)) < 0.0001)  l(i, j) = 0;  else  l(i, j) = (a(i, j) - sum) / d(j, j);  }  // Считаем диагональный элемент  {  real sum = 0;  for (int j = 0; j < i; ++j)  sum += d(j, j) \* l(i, j) \* l(i, j);  d(i, i) = a(i, i) - sum;  }  }  for (int i = 0; i < l.height(); i++)  l(i, i) = 1;    return true;  }  //-----------------------------------------------------------------------------  bool calcGaussianReverseOrder(const Matrix& l, const Matrix& y, Matrix& x) {  // l \* x = y, l - нижнетреугольная матрица  if (!l.isLowerTriangular() || !l.isDiagonalIdentity())  return false;  x.resize(1, y.height());  for (int i = x.height() - 1; i >= 0; --i) {  real sum = 0;  for (int j = i; j < x.height(); ++j)  sum += l(j, i) \* x(j, 0);  x(i, 0) = y(i, 0) - sum;  }  return true;  }  //-----------------------------------------------------------------------------  bool calcGaussianFrontOrder(const Matrix& l, const Matrix& y, Matrix& x) {  // l \* x = y, l - верхнетреугольная матрица  if (!l.isLowerTriangular() || !l.isDiagonalIdentity())  return false;  x.resize(1, y.height());  for (int i = 0; i < x.height(); ++i) {  real sum = 0;  for (int j = 0; j < i; ++j)  sum += l(i, j) \* x(j, 0);  x(i, 0) = y(i, 0) - sum;  }  return true;  }  //-----------------------------------------------------------------------------  bool calcGaussianCentralOrder(const Matrix& d, const Matrix& y, Matrix& x) {  // d \* x = y, d - диагональная матрица  if (!d.isDiagonal())  return false;  x.resize(1, y.height());  for (int i = 0; i < x.height(); ++i)  x(i, 0) = y(i, 0) / d(i, i);  return true;  }  //-----------------------------------------------------------------------------  bool solveSLAE\_by\_LDL(const Matrix& a, const Matrix& y, Matrix& x) {  // rus\_a \* x = y, rus\_a - симметричная матрица  if (!(a.width() == a.height() && a.width() == y.height() && !a.isDegenerate()))  return false;  Matrix l, d, z, w;  if (!calcLDL(a, l, d))  return false;  if (!calcGaussianFrontOrder(l, y, z))  return false;  if (!calcGaussianCentralOrder(d, z, w))  return false;  if (!calcGaussianReverseOrder(l, w, x))  return false;  return true;  }  //-----------------------------------------------------------------------------  bool solveSLAE\_byGaussMethod(const Matrix& a1, const Matrix& y1, Matrix& x1) {  if (!(a1.width() == a1.height() && y1.height() == a1.width() && !a1.isDegenerate()))  return false;  Matrix a(a1);  Matrix y(y1);  for (int i = 0; i < a.height(); ++i) {  // Находим максимальный элемент  int maxI = i;  for (int j = i+1; j < a.height(); ++j)  if (fabs(a(j, i)) > fabs(a(maxI, i)))  maxI = j;  // Переставляем эту строчку с текущей  for (int j = i; j < a.width(); ++j)  std::swap(a(i, j), a(maxI, j));  std::swap(y(i, 0), y(maxI, 0));  // Перебираем все строчки ниже и отнимаем текущую строчку от них  for (int j = i+1; j < a.height(); ++j) {  real m = a(j, i) / a(i, i);  for (int k = i; k < a.width(); ++k)  a(j, k) -= m \* a(i, k);  y(j, 0) -= m \* y(i, 0);  }  // Делим текущую строку на ее ведущий элемент, чтобы на диагонали были единицы  double m = a(i, i);  for (int j = i; j < a.width(); ++j)  a(i, j) /= m;  y(i, 0) /= m;  }  // Считаем обратный ход Гаусса  transpose(a);  calcGaussianReverseOrder(a, y, x1);  return true;  } |

**sparse.cpp:**

|  |
| --- |
| #include <fstream>  #include <iomanip>  #include <cmath>  #include <algorithm>  #include "sparse.h"  //-----------------------------------------------------------------------------  MatrixProfileSymmetric::MatrixProfileSymmetric() {  }  //-----------------------------------------------------------------------------  MatrixProfileSymmetric::MatrixProfileSymmetric(const Matrix& a) {  if (!a.isSymmetric())  throw std::exception();  di.resize(a.height(), 0);  ai.resize(a.height() + 1, 0);  for (int i = 0; i < a.height(); ++i) {  di[i] = a(i, i);  int zeroCount = 0;  for (; zeroCount < i; ++zeroCount)  if (a(i, zeroCount) != 0)  break;  int count = i - zeroCount;  ai[i+1] = ai[i] + count;  for (int j = 0; j < count; ++j)  al.push\_back(a(i, j + zeroCount));  }  }  //-----------------------------------------------------------------------------  void MatrixProfileSymmetric::loadFromFile(std::string fileName) {  std::ifstream fin(fileName);  int n;  fin >> n;  di.resize(n, 0);  for (int i = 0; i < n; ++i)  fin >> di[i];  fin >> n;  al.resize(n, 0);  for (int i = 0; i < n; ++i)  fin >> al[i];  fin >> n;  ai.resize(n, 0);  for (int i = 0; i < n; ++i)  fin >> ai[i];  fin.close();  }  //-----------------------------------------------------------------------------  void MatrixProfileSymmetric::saveToFile(std::string fileName) const {  std::ofstream fout(fileName);  fout.precision(std::numeric\_limits<real>::max\_digits10);  int w = std::numeric\_limits<real>::digits10 + 6;  fout << di.size() << std::endl;  for (const auto& i : di)  fout << std::setw(w) << i;  fout << std::endl;  fout << ai.size() << std::endl;  for (const auto& i : ai)  fout << std::setw(w) << i;  fout << std::endl;  fout << al.size() << std::endl;  for (const auto& i : al)  fout << std::setw(w) << i;  fout << std::endl;  fout.close();  }  //-----------------------------------------------------------------------------  void MatrixProfileSymmetric::toDenseMatrix(Matrix& dense) const {  if (dense.width() != size() && dense.height() != size())  dense.resize(size(), size());  for (int i = 0; i < size(); ++i) {  int iLineStart = getLineFirstElementPos(i);  int iLineSize = getLineSize(i);  for (int j = 0; j < iLineSize; ++j) {  dense(i, iLineStart + j) = getLineElement(i, j);  dense(iLineStart + j, i) = getLineElement(i, j);  }  dense(i, i) = getDiagonalElement(i);  }  }  //-----------------------------------------------------------------------------  void MatrixProfileSymmetric::negate(void) {  for (auto& i : al)  i = -i;  }  //-----------------------------------------------------------------------------  void MatrixProfileSymmetric::generate(int n, int min, int max, int maxDistanceToDiagonal) {  di.resize(n, 0);  ai.resize(n+1, 0);  al.clear();  al.reserve(maxDistanceToDiagonal \* n);  for (int i = 0; i < n; ++i) {  di[i] = intRandom(min, max);  int maxLineSize = i;  int lineSize = intRandom(0, std::min(maxLineSize, maxDistanceToDiagonal));  ai[i+1] = ai[i] + lineSize;  for (int j = 0; j < lineSize; ++j)  al.push\_back(intRandom(min, max));  }  }  //-----------------------------------------------------------------------------  int MatrixProfileSymmetric::size(void) const {  return di.size();  }  //-----------------------------------------------------------------------------  real& MatrixProfileSymmetric::getDiagonalElement(int n) {  return di[n];  }  //-----------------------------------------------------------------------------  const real& MatrixProfileSymmetric::getDiagonalElement(int n) const {  return di[n];  }  //-----------------------------------------------------------------------------  int MatrixProfileSymmetric::getLineFirstElementPos(int lineNo) const {  return lineNo - getLineSize(lineNo);  }  //-----------------------------------------------------------------------------  int MatrixProfileSymmetric::getLineSize(int lineNo) const {  return ai[lineNo+1] - ai[lineNo];  }  //-----------------------------------------------------------------------------  real& MatrixProfileSymmetric::getLineElement(int lineNo, int elemNo) {  return al[ai[lineNo] + elemNo];  }  //-----------------------------------------------------------------------------  const real& MatrixProfileSymmetric::getLineElement(int lineNo, int elemNo) const {  return al[ai[lineNo] + elemNo];  }  //-----------------------------------------------------------------------------  std::vector<real>::iterator MatrixProfileSymmetric::getLineFirstElement(int lineNo) {  return al.begin() + ai[lineNo];  }  //-----------------------------------------------------------------------------  std::vector<real>::const\_iterator MatrixProfileSymmetric::getLineFirstElement(int lineNo) const {  return al.begin() + ai[lineNo];  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  Vector::Vector() {  }  //-----------------------------------------------------------------------------  Vector::Vector(int size, real fill) : mas(size, 0) {  }  //-----------------------------------------------------------------------------  Vector::Vector(const Matrix& a) {  if (a.width() == 1) {  resize(a.height());  for (int i = 0; i < a.height(); ++i)  mas[i] = a(i, 0);  } else if (a.height() == 1) {  resize(a.width());  for (int i = 0; i < a.width(); ++i)  mas[i] = a(0, i);  } else  throw std::exception();  }  //-----------------------------------------------------------------------------  void Vector::toDenseMatrix(Matrix& dense, bool isVertical) const {  if (isVertical) {  dense.resize(1, mas.size());  for (int i = 0; i < mas.size(); ++i)  dense(i, 0) = mas[i];  } else {  dense.resize(mas.size(), 1);  for (int i = 0; i < mas.size(); ++i)  dense(0, i) = mas[i];  }  }  //-----------------------------------------------------------------------------  void Vector::loadFromFile(std::string fileName) {  std::ifstream fin(fileName);  int n;  fin >> n;  resize(n);  for (int i = 0; i < n; ++i)  fin >> mas[i];  fin.close();  }  //-----------------------------------------------------------------------------  void Vector::saveToFile(std::string fileName) const {  std::ofstream fout(fileName);  fout.precision(3);  fout << mas.size() << std::endl;  for (const auto& i : mas)  fout << std::scientific << i << std::endl;  fout.close();  }  //-----------------------------------------------------------------------------  void Vector::resize(int n, real fill) {  if (mas.size() != n)  mas.resize(n, fill);  }  //-----------------------------------------------------------------------------  void Vector::negate(void) {  for (auto& i : mas)  i = -i;  }  //-----------------------------------------------------------------------------  void Vector::generate(int n, int min, int max) {  resize(n);  for (int i = 0; i < n; ++i)  operator()(i) = intRandom(min, max);  }  //-----------------------------------------------------------------------------  void Vector::generate(int n) {  resize(n);  for (int i = 0; i < n; ++i)  operator()(i) = i+1;  }  //-----------------------------------------------------------------------------  int Vector::size(void) const {  return mas.size();  }  //-----------------------------------------------------------------------------  real& Vector::operator()(int i) {  return mas[i];  }  //-----------------------------------------------------------------------------  const real& Vector::operator()(int i) const {  return mas[i];  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  bool mul(const MatrixProfileSymmetric& a, const Vector& x, Vector& y) {  if (x.size() != a.size())  return false;  y.resize(x.size());  // Зануление результата  for (int i = 0; i < y.size(); ++i)  y(i) = 0;  // Умножение элементов из матрицы L  for (int i = 0; i < a.size(); ++i) {  int iLineStart = a.getLineFirstElementPos(i);  int iLineSize = a.getLineSize(i);  int j = iLineStart;  for (int k = 0; k < iLineSize; ++j, ++k) {  real elem = a.getLineElement(i, k);  y(i) += elem \* x(j);  y(j) += elem \* x(i);  }  }  // Умножение диагональных элементов на вектор  for (int i = 0; i < a.size(); ++i)  y(i) += a.getDiagonalElement(i) \* x(i);  return true;  }  //-----------------------------------------------------------------------------  bool sum(const Vector& a, const Vector& b, Vector& result) {  if (a.size() != b.size())  return false;  if (result.size() != a.size())  result.resize(a.size());  for (int i = 0; i < result.size(); ++i)  result(i) = a(i) + b(i);  return true;  }  //-----------------------------------------------------------------------------  sumreal sumAllElementsAbs(const Vector& a) {  sumreal sum = 0;  for (int i = 0; i < a.size(); ++i)  sum += fabs(a(i));  return sum;  }  //-----------------------------------------------------------------------------  void calcLDL(MatrixProfileSymmetric& a\_l) {  // Типо обращение к диагональному элементу  auto d = [&a\_l] (int i) -> real& {  return a\_l.getDiagonalElement(i);  };  // Перебираем все строки матрицы A  for (int i = 0; i < a\_l.size(); ++i) {  // Сумма для диагонального элемента  sumreal dsum = 0;  int iLineStart = a\_l.getLineFirstElementPos(i);  int iLineSize = a\_l.getLineSize(i);  // Считаем элементы матрицы L  for (int j = iLineStart; j < i; ++j) {  real& currentElem = a\_l.getLineElement(i, j - iLineStart);  int jLineStart = a\_l.getLineFirstElementPos(j);  int jLineSize = a\_l.getLineSize(j);  int offset = std::max(iLineStart, jLineStart);  int iLineOffset = offset - iLineStart;  int jLineOffset = offset - jLineStart;  int end = std::min(iLineStart + iLineSize, jLineStart + jLineSize);  int count = end - offset;  sumreal sum = 0;  for (int k = 0; k < count; ++k)  sum += d(offset + k) \*  a\_l.getLineElement(j, k + jLineOffset) \*  a\_l.getLineElement(i, k + iLineOffset);  currentElem = (currentElem - sum) / d(j);  dsum += d(j) \* currentElem \* currentElem;  }  // Считаем диагональный элемент  d(i) = d(i) - dsum;  }  }  //-----------------------------------------------------------------------------  void calcGaussianReverseOrder(const MatrixProfileSymmetric& a, Vector& y\_x) {  for (int i = y\_x.size() - 1; i >= 0; --i) {  int iLineStart = a.getLineFirstElementPos(i);  int iLineSize = a.getLineSize(i);  for (int j = iLineStart; j < i; j++)  y\_x(j) -= y\_x(i) \* a.getLineElement(i, j - iLineStart);  }  }  //-----------------------------------------------------------------------------  void calcGaussianFrontOrder(const MatrixProfileSymmetric& a, Vector& y\_x) {  for (int i = 0; i < y\_x.size(); ++i) {  int iLineStart = a.getLineFirstElementPos(i);  int iLineSize = a.getLineSize(i);  sumreal sum = 0;  for (int j = 0; j < iLineSize; ++j)  sum += a.getLineElement(i, j) \* y\_x(iLineStart + j);  y\_x(i) = y\_x(i) - sum;  }  }  //-----------------------------------------------------------------------------  void calcGaussianCentralOrder(const MatrixProfileSymmetric& d, Vector& y\_x) {  for (int i = 0; i < y\_x.size(); ++i)  y\_x(i) = y\_x(i) / d.getDiagonalElement(i);  // TODO что если элемент равен нулю?  }  //-----------------------------------------------------------------------------  void solveSLAE\_by\_LDL(MatrixProfileSymmetric& a, Vector& y\_x) {  y\_x.resize(a.size());  calcLDL(a);  calcGaussianFrontOrder(a, y\_x);  calcGaussianCentralOrder(a, y\_x);  calcGaussianReverseOrder(a, y\_x);  } |

**numerical\_tests.cpp:**

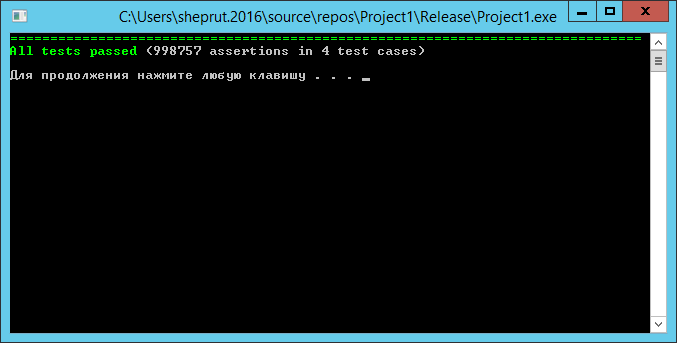
|  |
| --- |
| #include <sstream>  #include <cstdlib>  #include "matrix.h"  #include "sparse.h"  //-----------------------------------------------------------------------------  void test1(void) {  /\* 4-й пункт тестирования. \*/  Matrix a\_d;  generateTestMatrix(10, 4, a\_d);  a\_d.saveToFile("test1/a.txt");  Vector x;  x.generate(10);  Matrix x1, f1;  x.toDenseMatrix(x1, true);  mul(a\_d, x1, f1);  f1.saveToFile("test1/f.txt");  for (int k = 0; k < std::numeric\_limits<real>::digits10; ++k) {  MatrixProfileSymmetric a(a\_d);  a.getDiagonalElement(0) += pow(10.0, -k);  Vector f;  mul(a, x, f);  solveSLAE\_by\_LDL(a, f);  std::stringstream sout;  #ifdef ALL\_FLOAT  sout << "f";  #endif  #ifdef ALL\_DOUBLE  sout << "d";  #endif  #ifdef ALL\_FLOAT\_WITH\_DOUBLE  sout << "fd";  #endif  sout << "\_" << k+1 << ".txt";  Vector sub;  f.saveToFile("test1/result" + sout.str());  f.negate();  sum(x, f, sub);  sub.saveToFile("test1/sub" + sout.str());  }  }  //-----------------------------------------------------------------------------  void test2(void) {  /\* 5-й пункт тестирования. \*/  for (int k = 1; k < std::numeric\_limits<real>::digits10+2; ++k) {  Matrix a\_d;  generateGilbertMatrix(k, a\_d);  Vector x;  x.generate(k);  MatrixProfileSymmetric a(a\_d);  Vector f;  mul(a, x, f);  solveSLAE\_by\_LDL(a, f);  std::stringstream sout;  #ifdef ALL\_FLOAT  sout << "f";  #endif  #ifdef ALL\_DOUBLE  sout << "d";  #endif  #ifdef ALL\_FLOAT\_WITH\_DOUBLE  sout << "fd";  #endif  sout << "\_" << k << ".txt";  Vector sub;  f.saveToFile("test2/result" + sout.str());  f.negate();  sum(x, f, sub);  sub.saveToFile("test2/sub" + sout.str());  }  }  //-----------------------------------------------------------------------------  void test3(void) {  /\* 7-й пункт тестирования. \*/  for (int k = 1; k < std::numeric\_limits<real>::digits10+2; ++k) {  Matrix a;  generateGilbertMatrix(k, a);  Matrix x;  generateVector(k, x);  Matrix y;  mul(a, x, y);  Matrix x2;  solveSLAE\_byGaussMethod(a, y, x2);  std::stringstream sout;  #ifdef ALL\_FLOAT  sout << "f";  #endif  #ifdef ALL\_DOUBLE  sout << "d";  #endif  #ifdef ALL\_FLOAT\_WITH\_DOUBLE  sout << "fd";  #endif  sout << "\_" << k << ".txt";  Vector sub, x2\_s(x2), x\_s(x);  x2\_s.saveToFile("test3/result" + sout.str());  x2\_s.negate();  sum(x\_s, x2\_s, sub);  sub.saveToFile("test3/sub" + sout.str());  }  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  int main() {  test1();  test2();  test3();  } |

1. **Тесты**

Для тестирования использовалось юнит-тестирование и библиотека Catch. Было протестировано:

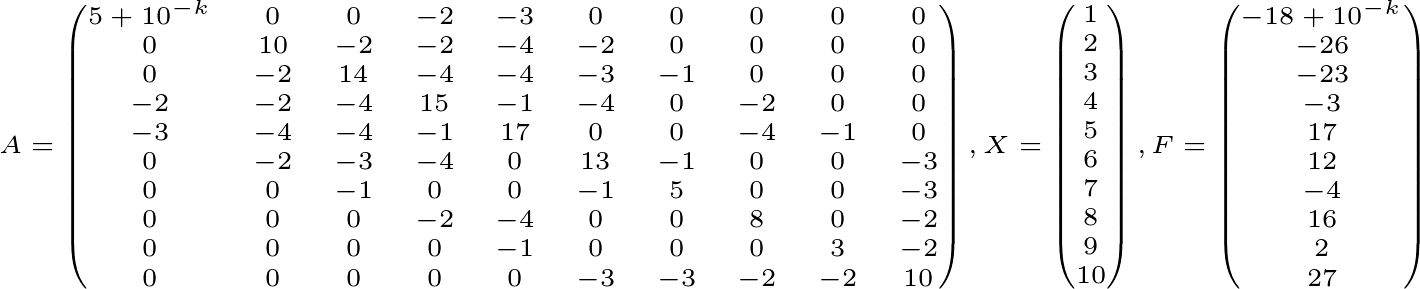
* Умножение профильной матрицы на вектор
* LDLT разложение
* Перевод матрицы из плотного формата и обратно
* Решение СЛАУ

Так же была протестирована работа программы на больших матрицах, размером 10 000 000, с максимальным размером профиля 5.



|  |
| --- |
| #define CATCH\_CONFIG\_RUNNER  #include "catch.hpp"  #include "matrix.h"  #include "sparse.h"  //-----------------------------------------------------------------------------  void testFormatByDenseMatrix(const Matrix& a\_d) {  // Общий тест перевода матрицы из плотного формата в разреженный  // Если плотная матрица не симметрична, то ее нельзя тестировать  if (!a\_d.isSymmetric())  throw std::exception();  // Переводим матрицу из плотного формата  MatrixProfileSymmetric a\_s(a\_d);    // Проверка размера  CHECK(a\_s.size() == a\_d.height());  // Проверка обратного преобразованя  Matrix sub, a\_d2;  a\_s.toDenseMatrix(a\_d2);  a\_d2.negate();  sum(a\_d, a\_d2, sub);  CHECK(isNear(sumAllElementsAbs(sub), 0));  //-------------------------------------------------------------------------  // Проверка формата  for (int i = 0; i < a\_d.height(); ++i) {  // Проверка диагональных элементов  CHECK(isNear(a\_s.getDiagonalElement(i), a\_d(i, i)));  // Определяем, с какого элемента начинается строка  int j = 0;  for (; j < i; ++j)  if (a\_d(i, j) != 0)  break;  // Проверка старта строки  CHECK(a\_s.getLineFirstElementPos(i) == j);  // Проверка размера строки  CHECK(a\_s.getLineSize(i) == (i - j));  // Проверка значений строки  for (int k = j; k < i; ++k)  CHECK(a\_s.getLineElement(i, k - j) == a\_d(i, k));  }  }  //-----------------------------------------------------------------------------  void testVectorByDenseMatrix(const Matrix& x\_d) {  // Общий тест перевода вектора из плотного формата в разреженный  // Если это не вектор, то проверить нельзя  if (x\_d.width() != 1)  throw std::exception();  // Переводим матрицу из плотного формата  Vector x\_s(x\_d);    // Проверка размера  CHECK(x\_s.size() == x\_d.height());  // Проверка обратного преобразованя  Matrix sub, x\_d2;  x\_s.toDenseMatrix(x\_d2, true);  x\_d2.negate();  sum(x\_d, x\_d2, sub);  CHECK(isNear(sumAllElementsAbs(sub), 0));  //-------------------------------------------------------------------------  // Проверка формата  for (int i = 0; i < x\_d.height(); ++i)  CHECK(isNear(x\_s(i), x\_d(i, 0)));  }  //-----------------------------------------------------------------------------  void testMultiplication(const Matrix& a\_d, const Matrix& b\_d) {  // Общий тест перемножения матриц в профильном формате через перемножение матриц в плотном формате  // Если матрица не является симметричной и ее нельзя перемножить на второй вектор, то тестировать это нельзя  if (!a\_d.isSymmetric() || b\_d.width() != 1 || b\_d.height() != a\_d.height())  throw std::exception();  MatrixProfileSymmetric a\_s(a\_d);  Vector b\_s(b\_d);  // Перемножаем исходные матрицы в плотном формате  Matrix c\_d;  mul(a\_d, b\_d, c\_d);  // Перемножаем исходные матрицы в профильном формате  Vector c\_s;  mul(a\_s, b\_s, c\_s);  // Преобразуем полученную матрицу в плотный формат  Matrix c\_d2;  c\_s.toDenseMatrix(c\_d2, true);  // Находим разность между ними  Matrix sub;  c\_d2.negate();  sum(c\_d, c\_d2, sub);  // Она должна быть близка нулю  CHECK(isNear(sumAllElementsAbs(sub), 0));  }  //-----------------------------------------------------------------------------  void testLDL(const Matrix& l\_d, const Matrix& d\_d) {  // Общий тест LDL^T разложения путем перемножения матриц L и D, и сравнения их с результатом разложения  // Если матрицы L и D не являются теми, кем должны являться, то проверять нельзя  if (!l\_d.isLowerTriangular() || !l\_d.isDiagonalIdentity() || !d\_d.isDiagonal())  throw std::exception();  // Получаем транспонированную матрицу в плотном формате  Matrix lt\_d = l\_d;  transpose(lt\_d);  // a = l \* d \* l^T  Matrix a\_d, ld\_d;  mul(l\_d, d\_d, ld\_d);  mul(ld\_d, lt\_d, a\_d);  // Считаем LDL^T разложение в плотном формате  MatrixProfileSymmetric a\_s(a\_d);  calcLDL(a\_s);  // Проверяем полученные данные с имеющимися матрицами  for (int i = 0; i < a\_s.size(); ++i) {  // Проверяем диагональные элементы  CHECK(isNear(a\_s.getDiagonalElement(i), d\_d(i, i)));  // Проверяем L  for (int j = 0; j < a\_s.getLineSize(i); ++j)  CHECK(isNear(  a\_s.getLineElement(i, j),  l\_d(i, j + a\_s.getLineFirstElementPos(i))  ));  }  }  //-----------------------------------------------------------------------------  void testSolve(const Matrix& a\_d, const Matrix& x\_d) {  // Проверка решения полного цикла СЛАУ через умножение матриц  if (!a\_d.isSymmetric() || x\_d.width() != 1 || x\_d.height() != a\_d.height())  throw std::exception();  // Генерируем матрицу и вектор заданного размера  MatrixProfileSymmetric a\_s(a\_d);  Vector x\_s(x\_d);  // Получаем вектор: y = a \* x  Vector y\_x;  mul(a\_s, x\_s, y\_x);  // Решаем СЛАУ  solveSLAE\_by\_LDL(a\_s, y\_x);  y\_x.negate(); // y\_x = x\*  // Отнимаем полученное решение от изначально сгенерированного вектора  Matrix sub, x2\_d;  y\_x.toDenseMatrix(x2\_d, true);  sum(x2\_d, x\_d, sub); // sub = x - x\*  // Разность решения и изначального вектора должна быть близка к нулю  CHECK(isNear(sumAllElementsAbs(sub), 0));  }  //-----------------------------------------------------------------------------  void testSolveOnBigData(int size, int maxDistanceToDiagonal) {  // Проверка решения полного цикла СЛАУ через умножение случайно сгенерированных матриц  // Генерируем матрицу и вектор заданного размера  MatrixProfileSymmetric a;  Vector x;  a.generate(size, 1, 20, maxDistanceToDiagonal);  x.generate(5, 1, 20);  // Получаем вектор: y = a \* x  Vector y\_x;  mul(a, x, y\_x);  // Решаем СЛАУ  solveSLAE\_by\_LDL(a, y\_x);  y\_x.negate(); // y\_x = x\*  // Отнимаем полученное решение от изначально сгенерированного вектора  Matrix sub, x2\_d, x\_d;  y\_x.toDenseMatrix(x2\_d, true);  x.toDenseMatrix(x\_d, true);  sum(x2\_d, x\_d, sub); // sub = x - x\*  // Разность решения и изначального вектора должна быть близка к нулю  CHECK(isNear(sumAllElementsAbs(sub), 0));  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  TEST\_CASE("Dense converting test") {  Matrix a;  a.getFromVector(5, 5, {  1, 0, 2, 0, 0,  0, 3, 0, 0, 0,  2, 0, -1, 2, 4,  0, 0, 2, 2.2,0,  0, 0, 4, 0, 5,  });  testFormatByDenseMatrix(a);  a.getFromVector(6, 6, {  4, 1, 1, 0, 0, 0,  1, 9, 6, 0, 0, 0,  1, 6, 16, 0, 4, 0,  0, 0, 0, 99, -1, 0,  0, 0, 4, -1, 100,100,  0, 0, 0, 0, 100,-99,  });  testFormatByDenseMatrix(a);  generateSparseSymmetricMatrix(10, 0, 100, 0.5, a);  testFormatByDenseMatrix(a);  generateSparseSymmetricMatrix(100, 0, 100, 0.5, a);  testFormatByDenseMatrix(a);  generateSparseSymmetricMatrix(1000, 0, 100, 0.1, a);  testFormatByDenseMatrix(a);  //-------------------------------------------------------------------------  Matrix x;  x.getFromVector(1, 5, {  0, 2, 3, 4, 5  });  testVectorByDenseMatrix(x);  x.getFromVector(1, 6, {  0, 2, 3, 4, 5, 0  });  testVectorByDenseMatrix(x);  generateVector(10, 0, 100, x);  testVectorByDenseMatrix(x);  generateVector(100, 0, 100, x);  testVectorByDenseMatrix(x);  generateVector(100, 0, 100, x);  testVectorByDenseMatrix(x);  }  //-----------------------------------------------------------------------------  TEST\_CASE("Multiplication test") {  Matrix a, x;  a.getFromVector(5, 5, {  1, 0, 2, 0, 0,  0, 3, 0, 0, 0,  2, 0, -1, 2, 4,  0, 0, 2, 2.2,0,  0, 0, 4, 0, 5,  });  x.getFromVector(1, 5, {  1, 2, 3, 4, 5  });  testMultiplication(a, x);  a.getFromVector(6, 6, {  4, 1, 1, 0, 0, 0,  1, 9, 6, 0, 0, 0,  1, 6, 16, 0, 4, 0,  0, 0, 0, 99, -1, 0,  0, 0, 4, -1, 100,100,  0, 0, 0, 0, 100,-99,  });  x.getFromVector(1, 6, {  1, 2, 3, 4, 5, 6  });  testMultiplication(a, x);  generateSparseSymmetricMatrix(10, 0, 100, 0.5, a);  generateVector(10, 0, 100, x);  testMultiplication(a, x);  generateSparseSymmetricMatrix(100, 0, 100, 0.5, a);  generateVector(100, 0, 100, x);  testMultiplication(a, x);  generateSparseSymmetricMatrix(1000, 0, 100, 0.1, a);  generateVector(1000, 0, 100, x);  testMultiplication(a, x);  }  //-----------------------------------------------------------------------------  TEST\_CASE("LDL test") {  Matrix l, d;  l.getFromVector(5, 5, {  1, 0, 0, 0, 0,  0, 1, 0, 0, 0,  0, 2, 1, 0, 0,  0, 3, -1, 1, 0,  0, 0, 0, 5, 1,  });  d.getFromVector(5, 5, {  1, 0, 0, 0, 0,  0, 6, 0, 0, 0,  0, 0, 10, 0, 0,  0, 0, 0, 1, 0,  0, 0, 0, 0, -1,  });  testLDL(l, d);  l.getFromVector(6, 6, {  1, 0, 0, 0, 0, 0,  2, 1, 0, 0, 0, 0,  1, 0, 1, 0, 0, 0,  0, 0, 0, 1, 0, 0,  0, 0, -9, 5, 1, 0,  1, 2, 3, 4, 5, 1,  });  d.getFromVector(6, 6, {  10, 0, 0, 0, 0, 0,  0, 9, 0, 0, 0, 0,  0, 0, 8, 0, 0, 0,  0, 0, 0, 7, 0, 0,  0, 0, 0, 0, 6, 0,  0, 0, 0, 0, 0, 5,  });  testLDL(l, d);  generateLMatrix(10, 0, 100, 0.5, l);  generateDiagonalMatrix(10, 1, 100, d);  testLDL(l, d);  generateLMatrix(100, 0, 100, 0.5, l);  generateDiagonalMatrix(100, 1, 100, d);  testLDL(l, d);  generateLMatrix(1000, 0, 100, 0.1, l);  generateDiagonalMatrix(1000, 1, 100, d);  testLDL(l, d);  }  //-----------------------------------------------------------------------------  TEST\_CASE("Random matrixes") {  Matrix a, x;  a.getFromVector(5, 5, {  1, 0, 2, 0, 0,  0, 3, 0, 0, 0,  2, 0, -1, 2, 4,  0, 0, 2, 2.2,0,  0, 0, 4, 0, 5,  });  x.getFromVector(1, 5, {  1, 2, 3, 4, 5  });  testMultiplication(a, x);  a.getFromVector(6, 6, {  4, 1, 1, 0, 0, 0,  1, 9, 6, 0, 0, 0,  1, 6, 16, 0, 4, 0,  0, 0, 0, 99, -1, 0,  0, 0, 4, -1, 100,100,  0, 0, 0, 0, 100,-99,  });  x.getFromVector(1, 6, {  1, 2, 3, 4, 5, 6  });  testMultiplication(a, x);  testSolveOnBigData(5000, 50);  testSolveOnBigData(50000, 50);  testSolveOnBigData(500000, 50);  testSolveOnBigData(5000000, 5);  testSolveOnBigData(10000000, 5);  }  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  //-----------------------------------------------------------------------------  int main(int argc, char\* const argv[]) {  int result = Catch::Session().run(argc, argv);  system("pause");  return result;  } |

1. **Оценка влияния увеличения числа обусловленности на точность решения**



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| k | xk  **(одинарная точность)** | x\*-xk  **(одинарная точность)** | xk  **(двойная точность)** | x\*-xk  **(двойная точность)** | xk  **(смешанная точность)** | x\*-xk  **(смешанная точность)** |
| 1 | 1.00001  2.00001  3.00001  4.00001  5.00001  6.00001  7.00001  8.00001  9.00001  10 | -7.391e-06  -9.060e-06  -9.060e-06  -9.060e-06  -8.583e-06  -9.060e-06  -1.001e-05  -9.537e-06  -9.537e-06  -1.049e-05 | 0.999999999999992  1.99999999999999  2.99999999999999  3.99999999999999  4.99999999999999  5.99999999999999  6.99999999999999  7.99999999999999  8.99999999999999  9.99999999999999 | 8.438e-15  9.770e-15  9.770e-15  1.021e-14  9.770e-15  1.066e-14  8.882e-15  1.066e-14  8.882e-15  1.066e-14 | 0.999994  1.99999  2.99999  3.99999  4.99999  5.99999  6.99999  7.99999  8.99999  9.99999 | 6.199e-06  7.391e-06  7.868e-06  7.391e-06  7.629e-06  7.629e-06  7.629e-06  7.629e-06  7.629e-06  7.629e-06 |
| 2 | 1  2  3  4  5  6  7  8  9  10 | -8.345e-07  -9.537e-07  -4.768e-07  -4.768e-07  -9.537e-07  0.000e+00  -4.768e-07  -9.537e-07  0.000e+00  0.000e+00 | 1  2  3  4  5  6  7  8  9  10 | 2.220e-16  2.220e-16  8.882e-16  4.441e-16  8.882e-16  0.000e+00  8.882e-16  8.882e-16  0.000e+00  0.000e+00 | 1.00006  2.00006  3.00006  4.00006  5.00006  6.00006  7.00006  8.00006  9.00006  10.0001 | -6.258e-05  -6.413e-05  -6.390e-05  -6.390e-05  -6.390e-05  -6.437e-05  -6.437e-05  -6.485e-05  -6.390e-05  -6.485e-05 |
| 3 | 0.999619  1.99962  2.99962  3.99962  4.99962  5.99962  6.99962  7.99962  8.99962  9.99962 | 3.806e-04  3.817e-04  3.817e-04  3.815e-04  3.815e-04  3.824e-04  3.824e-04  3.824e-04  3.824e-04  3.834e-04 | 1.00000000000142  2.00000000000143  3.00000000000143  4.00000000000143  5.00000000000142  6.00000000000143  7.00000000000143  8.00000000000142  9.00000000000142  10.0000000000014 | -1.422e-12  -1.426e-12  -1.426e-12  -1.426e-12  -1.425e-12  -1.426e-12  -1.426e-12  -1.425e-12  -1.425e-12  -1.426e-12 | 1.00031  2.00031  3.00031  4.00031  5.00031  6.00031  7.00031  8.00031  9.00031  10.0003 | -3.098e-04  -3.104e-04  -3.104e-04  -3.109e-04  -3.099e-04  -3.109e-04  -3.119e-04  -3.099e-04  -3.109e-04  -3.109e-04 |
| 4 | 0.996184  1.99618  2.99618  3.99618  4.99618  5.99618  6.99618  7.99618  8.99618  9.99618 | 3.816e-03  3.816e-03  3.817e-03  3.816e-03  3.817e-03  3.817e-03  3.816e-03  3.816e-03  3.817e-03  3.817e-03 | 1.00000000001776  2.00000000001777  3.00000000001777  4.00000000001777  5.00000000001777  6.00000000001777  7.00000000001777  8.00000000001777  9.00000000001777  10.0000000000178 | -1.776e-11  -1.777e-11  -1.777e-11  -1.777e-11  -1.777e-11  -1.777e-11  -1.777e-11  -1.777e-11  -1.777e-11  -1.777e-11 | 1.00286  2.00286  3.00286  4.00286  5.00286  6.00286  7.00286  8.00286  9.00286  10.0029 | -2.863e-03  -2.864e-03  -2.863e-03  -2.864e-03  -2.864e-03  -2.864e-03  -2.864e-03  -2.864e-03  -2.863e-03  -2.864e-03 |
| 5 | 0.759266  1.75926  2.75926  3.75926  4.75926  5.75926  6.75926  7.75926  8.75926  9.75926 | 2.407e-01  2.407e-01  2.407e-01  2.407e-01  2.407e-01  2.407e-01  2.407e-01  2.407e-01  2.407e-01  2.407e-01 | 1.00000000028422  2.00000000028423  3.00000000028423  4.00000000028423  5.00000000028422  6.00000000028423  7.00000000028423  8.00000000028423  9.00000000028423  10.0000000002842 | -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10  -2.842e-10 | 0.941177  1.94118  2.94118  3.94118  4.94118  5.94118  6.94118  7.94118  8.94118  9.94118 | 5.882e-02  5.882e-02  5.882e-02  5.882e-02  5.882e-02  5.882e-02  5.882e-02  5.882e-02  5.882e-02  5.882e-02 |
| 6 | 1.66666  2.66667  3.66667  4.66667  5.66667  6.66667  7.66667  8.66667  9.66667  10.6667 | -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01  -6.667e-01 | 0.999999998934186  1.99999999893418  2.99999999893418  3.99999999893418  4.99999999893418  5.99999999893418  6.99999999893418  7.99999999893418  8.99999999893418  9.99999999893418 | 1.066e-09  1.066e-09  1.066e-09  1.066e-09  1.066e-09  1.066e-09  1.066e-09  1.066e-09  1.066e-09  1.066e-09 | 1.90735e-06  1  2  3  4  5  6  7  8  9 | 1.000e+00  1.000e+00  1.000e+00  1.000e+00  1.000e+00  1.000e+00  1.000e+00  1.000e+00  1.000e+00  1.000e+00 |
| 7 |  |  | 1.00000002842171  2.00000002842172  3.00000002842172  4.00000002842172  5.00000002842172  6.00000002842172  7.00000002842172  8.00000002842172  9.00000002842172  10.0000000284217 | -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08  -2.842e-08 |  |  |
| 8 |  |  | 1.00000007105427  2.00000007105428  3.00000007105428  4.00000007105428  5.00000007105428  6.00000007105428  7.00000007105428  8.00000007105428  9.00000007105428  10.0000000710543 | -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08  -7.105e-08 |  |  |
| 9 |  |  | 1.0000003552714  2.0000003552714  3.0000003552714  4.0000003552714  5.0000003552714  6.0000003552714  7.0000003552714  8.0000003552714  9.0000003552714  10.0000003552714 | -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07  -3.553e-07 |  |  |
| 10 |  |  | 1.00002131635604  2.00002131635604  3.00002131635604  4.00002131635604  5.00002131635604  6.00002131635604  7.00002131635604  8.00002131635604  9.00002131635604  10.000021316356 | -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05  -2.132e-05 |  |  |
| 11 |  |  | 1.00028422716856  2.00028422716856  3.00028422716856  4.00028422716856  5.00028422716856  6.00028422716856  7.00028422716856  8.00028422716856  9.00028422716856  10.0002842271686 | -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04  -2.842e-04 |  |  |
| 12 |  |  | 0.999644760213144  1.99964476021314  2.99964476021314  3.99964476021314  4.99964476021314  5.99964476021314  6.99964476021314  7.99964476021314  8.99964476021314  9.99964476021314 | 3.552e-04  3.552e-04  3.552e-04  3.552e-04  3.552e-04  3.552e-04  3.552e-04  3.552e-04  3.552e-04  3.552e-04 |  |  |
| 13 |  |  | 0.985815602836885  1.98581560283688  2.98581560283688  3.98581560283688  4.98581560283688  5.98581560283688  6.98581560283688  7.98581560283688  8.98581560283688  9.98581560283688 | 1.418e-02  1.418e-02  1.418e-02  1.418e-02  1.418e-02  1.418e-02  1.418e-02  1.418e-02  1.418e-02  1.418e-02 |  |  |
| 14 |  |  | 1.22222222222222  2.22222222222222  3.22222222222222  4.22222222222222  5.22222222222222  6.22222222222222  7.22222222222222  8.22222222222222  9.22222222222222  10.2222222222222 | -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01  -2.222e-01 |  |  |

**Выводы:**

1. Использование типа double для подсчета скалярного произведения дает результат немного точнее, чем при использовании float.
2. Двойная точность точнее на 10 порядков, чем одинарная.
3. С увеличением k падает точность, так как все элементы зависят от первого элемента.
4. **Исследование на матрицах Гильберта**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| k | xk  **(одинарная точность)** | x\*-xk  **(одинарная точность)** | xk  **(двойная точность)** | x\*-xk  **(двойная точность)** | xk  **(смешанная точность)** | x\*-xk  **(смешанная точность)** |
| 1 | 1 | 0.000e+00 | 1 | 0.000e+00 | 1 | 0.000e+00 |
| 2 | 1  2 | 3.576e-07  -7.153e-07 | 1  2 | -6.661e-16  1.332e-15 | 1  2 | 3.576e-07  -7.153e-07 |
| 3 | 1  1.99999  3.00001 | -1.311e-06  9.775e-06  -1.073e-05 | 1  1.99999999999998  3.00000000000002 | -3.775e-15  1.732e-14  -1.510e-14 | 1  2  3.00001 | -3.576e-07  4.292e-06  -5.245e-06 |
| 4 | 0.999971  2.00032  2.99923  4.0005 | 2.861e-05  -3.223e-04  7.730e-04  -5.007e-04 | 1.00000000000015  1.99999999999839  3.00000000000384  3.99999999999751 | -1.459e-13  1.612e-12  -3.845e-12  2.487e-12 | 0.999965  2.00039  2.99907  4.00061 | 3.505e-05  -3.905e-04  9.346e-04  -6.051e-04 |
| 5 | 0.999974  2.00057  2.99743  4.00397  4.99804 | 2.646e-05  -5.670e-04  2.571e-03  -3.971e-03  1.965e-03 | 1.00000000000114  1.99999999997896  3.00000000009007  3.99999999986454  5.0000000000661 | -1.142e-12  2.104e-11  -9.007e-11  1.355e-10  -6.610e-11 | 0.99999  2.00025  2.99878  4.00195  4.99902 | 1.001e-05  -2.532e-04  1.219e-03  -1.946e-03  9.847e-04 |
| 6 |  |  | 0.999999999999456  2.00000000002036  2.99999999984237  4.00000000044523  4.99999999948028  6.00000000021327 | 5.442e-13  -2.036e-11  1.576e-10  -4.452e-10  5.197e-10  -2.133e-10 |  |  |
| 7 |  |  | 0.999999999970605  2.00000000119635  2.99999998831331  4.00000004590216  4.99999991518355  6.00000007374735  6.99999997566314 | 2.940e-11  -1.196e-09  1.169e-08  -4.590e-08  8.482e-08  -7.375e-08  2.434e-08 |  |  |
| 8 |  |  | 0.999999999770664  2.00000001221339  2.99999984135592  4.00000085494724  4.99999770598508  6.00000323711437  6.99999770146461  8.00000064730651 | 2.293e-10  -1.221e-08  1.586e-07  -8.549e-07  2.294e-06  -3.237e-06  2.299e-06  -6.473e-07 |  |  |
| 9 |  |  | 0.999999999404365  2.00000003862404  2.9999993763298  4.00000429057047  4.99998472555468  6.00003042841748  6.99996577033999  8.0000203137782  8.99999505671733 | 5.956e-10  -3.862e-08  6.237e-07  -4.291e-06  1.527e-05  -3.043e-05  3.423e-05  -2.031e-05  4.943e-06 |  |  |
| 10 |  |  | 0.999999984955016  2.00000127747496  2.99997315256078  4.00024148041821  4.99885817958898  6.00311612995441  6.99491877971188  8.00488449114173  8.9974474491523  10.0005590841989 | 1.504e-08  -1.277e-06  2.685e-05  -2.415e-04  1.142e-03  -3.116e-03  5.081e-03  -4.884e-03  2.553e-03  -5.591e-04 |  |  |
| 11 |  |  | 1.00000002832744  1.99999696547234  3.00008018508687  3.99908988554279  5.00549049531512  5.98048984710493  7.04286901053999  7.94109396943942  9.049269492832  9.97706497978577  11.0045551622683 | -2.833e-08  3.035e-06  -8.019e-05  9.101e-04  -5.490e-03  1.951e-02  -4.287e-02  5.891e-02  -4.927e-02  2.294e-02  -4.555e-03 |  |  |
| 12 |  |  | 0.999999938707628  2.00000799892841  2.99974322311247  4.00355113702771  4.97367607094356  6.11664500758792  6.67285746001541  8.59523157049891  8.29929894188331  10.5148722155977  10.7853593823423  12.0387570998998 | 6.129e-08  -7.999e-06  2.568e-04  -3.551e-03  2.632e-02  -1.166e-01  3.271e-01  -5.952e-01  7.007e-01  -5.149e-01  2.146e-01  -3.876e-02 |  |  |

**Выводы:**

1. На этот раз использование double для подсчета скалярного произведения не дало существенного увеличения точности.
2. С ростом размера матрицы Гильберта падала точность.
3. **Метод Гаусса**

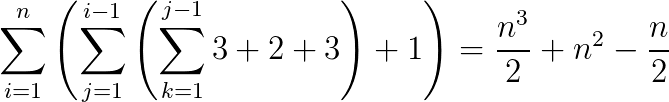
Метод Гаусса считался для матриц Гильберта. Был реализован Метод Гаусса с выбором ведущего элемента.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| k | x\*-xk  **(одинарная точность)**  **(LDL)** | x\*-xk  **(одинарная точность)**  **(Гаусс)** | x\*-xk  **(двойная точность)**  **(LDL)** | x\*-xk  **(двойная точность)**  **(Гаусс)** |
| 1 | 0.000e+00 | 0.000e+00 | 0.000e+00 | 0.000e+00 |
| 2 | 3.576e-07  -7.153e-07 | 3.576e-07  -7.153e-07 | -6.661e-16  1.332e-15 | -6.661e-16  1.332e-15 |
| 3 | -1.311e-06  9.775e-06  -1.073e-05 | -3.576e-07  4.292e-06  -5.245e-06 | -3.775e-15  1.732e-14  -1.510e-14 | -2.442e-15  1.155e-14  -1.021e-14 |
| 4 | 2.861e-05  -3.223e-04  7.730e-04  -5.007e-04 | -2.193e-05  2.499e-04  -6.022e-04  3.912e-04 | -1.459e-13  1.612e-12  -3.845e-12  2.487e-12 | -3.464e-14  4.112e-13  -1.015e-12  6.701e-13 |
| 5 | 2.646e-05  -5.670e-04  2.571e-03  -3.971e-03  1.965e-03 | 1.097e-05  -6.676e-05  -3.624e-05  3.595e-04  -2.770e-04 | -1.142e-12  2.104e-11  -9.007e-11  1.355e-10  -6.610e-11 | 6.040e-14  -1.112e-12  4.693e-12  -6.953e-12  3.347e-12 |
| 6 |  |  | 5.442e-13  -2.036e-11  1.576e-10  -4.452e-10  5.197e-10  -2.133e-10 | 5.294e-13  -1.547e-11  1.045e-10  -2.698e-10  2.953e-10  -1.155e-10 |
| 7 |  |  | 2.940e-11  -1.196e-09  1.169e-08  -4.590e-08  8.482e-08  -7.375e-08  2.434e-08 | 1.924e-12  -7.008e-11  6.235e-10  -2.266e-09  3.926e-09  -3.232e-09  1.018e-09 |
| 8 |  |  | 2.293e-10  -1.221e-08  1.586e-07  -8.549e-07  2.294e-06  -3.237e-06  2.299e-06  -6.473e-07 | -2.357e-11  1.236e-09  -1.580e-08  8.395e-08  -2.224e-07  3.102e-07  -2.180e-07  6.083e-08 |
| 9 |  |  | 5.956e-10  -3.862e-08  6.237e-07  -4.291e-06  1.527e-05  -3.043e-05  3.423e-05  -2.031e-05  4.943e-06 | -7.835e-10  5.365e-08  -9.020e-07  6.403e-06  -2.337e-05  4.752e-05  -5.438e-05  3.274e-05  -8.067e-06 |
| 10 |  |  | 1.504e-08  -1.277e-06  2.685e-05  -2.415e-04  1.142e-03  -3.116e-03  5.081e-03  -4.884e-03  2.553e-03  -5.591e-04 | -5.034e-09  4.377e-07  -9.365e-06  8.543e-05  -4.085e-04  1.125e-03  -1.849e-03  1.789e-03  -9.403e-04  2.070e-04 |
| 11 |  |  | -2.833e-08  3.035e-06  -8.019e-05  9.101e-04  -5.490e-03  1.951e-02  -4.287e-02  5.891e-02  -4.927e-02  2.294e-02  -4.555e-03 | -1.190e-08  1.160e-06  -2.822e-05  2.976e-04  -1.681e-03  5.630e-03  -1.172e-02  1.533e-02  -1.226e-02  5.474e-03  -1.046e-03 |
| 12 |  |  | 6.129e-08  -7.999e-06  2.568e-04  -3.551e-03  2.632e-02  -1.166e-01  3.271e-01  -5.952e-01  7.007e-01  -5.149e-01  2.146e-01  -3.876e-02 | -1.365e-07  1.717e-05  -5.367e-04  7.278e-03  -5.314e-02  2.327e-01  -6.464e-01  1.167e+00  -1.365e+00  9.979e-01  -4.142e-01  7.450e-02 |

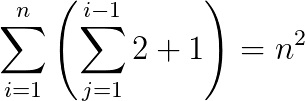
**Выводы:**

1. Для одинарной точности метод Гаусса чуть точнее метода решения СЛАУ через разложение LDLT.
2. Для двойной точности метод Гаусса до k=9 был на порядок точнее, чем метод решения СЛАУ через разложение LDLT, а затем они сравнялись в точности.
3. **Расчет количества действий**

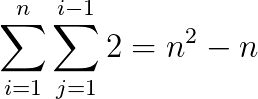
* **Разложение LDLT:**



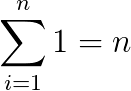
* **Прямой обход:**



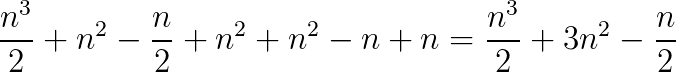
* **Обратный обход:**



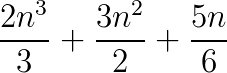
* **Центральный обход:**



* **Всего:**



* **Метод Гаусса:**

****