**Московский авиационный институт**

**(Национальный исследовательский университет)**

**Факультет: «Информационные технологии и прикладная математика»**

**Кафедра: 806 «Вычислительная математика и программирование»**

**Дисциплина: «Численные методы»**

**Лабораторная работы по курсу**

**“Численные методы”**

Студент: Семенов И. М.

Группа: М8О-306Б-18

Преподаватель: Ревизников Д. Л.

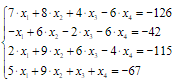
Дата: 5.06.2021

Оценка:

Москва, 2021

# Лабораторная работа 1.1

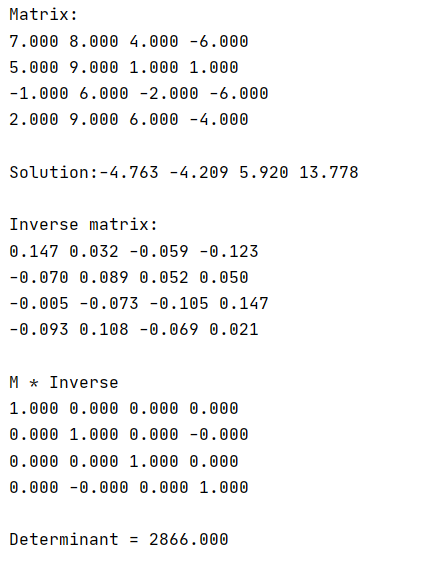
Задание: Реализовать алгоритм LU – разложения матриц(с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений. Для матрицы СЛАУ вычислить определитель и обратную матрицу

Условие:

Исходный код: matrix.h

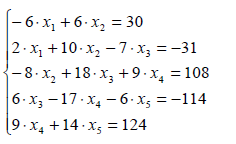
#ifndef **LAB1\_MATRIX\_H**#define **LAB1\_MATRIX\_H**#include <vector>  
#include <stdexcept>  
#include <iostream>  
#include <tuple>  
#include <complex>  
#include <functional>  
  
template <typename T>  
using Column = std::vector<T>;  
  
template <typename T>  
std::ostream& operator << (std::ostream& os, const std::vector<T>& v) {  
 for (const auto& i : v) {  
 os << i << " ";  
 }  
 return os;  
}  
  
template <typename T>  
std::ostream& operator << (std::ostream& os, std::complex<T> cp) {  
 return os << "(re: " << cp.real() << ", im:" << cp.imag() << ")";  
}  
  
template <typename T>  
Column<T> operator \* (Column<T> v, T value) {  
 for (T& i : v) {  
 i \*= value;  
 }  
 return v;  
}  
  
template <typename T>  
Column<T> operator \* (T value, Column<T> v) {  
 return std::move(v) \* value;  
}  
  
template <typename T>  
Column<T> operator / (Column<T> v, T value) {  
 for (T& i : v) {  
 i /= value;  
 }  
 return v;  
}  
  
template <typename T>  
Column<T> operator / (T value, Column<T> v) {  
 return std::move(v) / value;  
}  
  
template <typename T>  
class Matrix {  
public:  
 Matrix(const Matrix<T>&);  
 Matrix(Matrix<T>&&);  
  
 Matrix& operator = (const Matrix<T>&other);  
 Matrix& operator = (Matrix<T>&&other);  
  
 Matrix(std::vector<std::vector<T>>);  
  
 static Matrix<T> ones(int n, int m);  
 static Matrix<T> zeros(int n, int m);  
  
 static Matrix<T> construct\_row\_matrix(std::vector<T>);  
 static Matrix<T> construct\_row\_matrix(size\_t);  
  
 static Matrix<T> construct\_column\_matrix(const std::vector<T>&);  
 static Matrix<T> construct\_column\_matrix(size\_t);  
  
 static Matrix<T> concatenate\_columns(const std::vector<Column<T>>&);  
  
 auto begin();  
 auto begin() const;  
 auto end();  
 auto end() const;  
  
  
 Matrix<T> operator \* (T other) const;  
 Matrix<T> operator \* (const Matrix<T>& other) const;  
 Matrix<T> operator \* (const Column<T>& other) const;  
  
 Matrix<T> operator + (const Matrix<T>& other) const;  
 Matrix<T> operator - (const Matrix<T>& other) const;  
  
  
  
 Matrix<T> transpose() const;  
 const std::vector<std::vector<T>>& get\_data() const;  
  
 Column<T> get\_column(size\_t index) const;  
 explicit operator Column<T>() const;  
 explicit operator T() const;  
 std::vector<T>& operator[](size\_t index);  
 const std::vector<T>& operator[](size\_t index) const;  
  
 size\_t size\_rows() const;  
 size\_t size\_columns() const;  
  
 void swap\_rows(int f, int s);  
 void swap\_columns(int f, int s);  
  
 std::vector<std::vector<T>> data;  
};  
  
  
template <typename T>  
struct LUDecomposition {  
 using value\_type = T;  
 Matrix<T> L;  
 Matrix<T> U;  
 Matrix<T> P;  
 bool odd;  
};  
  
template <typename T>  
std::ostream& operator << (std::ostream& os, const Matrix<T>& m) {  
 for (int i = 0; i < m.size\_rows(); ++i) {  
 for (int j = 0; j < m.size\_columns(); ++j) {  
 os << m[i][j] << " ";  
 }  
 os << "\n";  
 }  
 return os;  
}  
  
template <typename T>  
Matrix<T>::operator T() const {  
 if (size\_rows() == 1 && size\_columns() == 1) {  
 return (\*this)[0][0];  
 }  
 throw std::logic\_error("cant cast matrix to number");  
}  
  
template<typename T>  
Matrix<T> operator \* (Matrix<T> mt, T value) {  
 for (int i = 0; mt.size\_rows(); ++i) {  
 for (int j = 0; mt.size\_columns(); ++i) {  
 mt[i][j] \*= value;  
 }  
 }  
 return mt;  
}  
  
template<typename T>  
Matrix<T> operator \* (T value, Matrix<T> mt) {  
 return mt \* value;  
}  
  
  
template <typename T>  
Matrix<T>::Matrix(const Matrix<T>& other)  
: data(other.data) {}  
  
template <typename T>  
Matrix<T>::Matrix(Matrix<T>&& other)  
: data(std::move(other.data)) {}  
  
template <typename T>  
Matrix<T>::Matrix(std::vector<std::vector<T>> v)  
: data(std::move(v)) {}  
  
template <typename T>  
const std::vector<std::vector<T>>& Matrix<T>::get\_data() const {  
 return data;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::zeros(int n, int m) {  
 return std::vector<std::vector<T>>(n, std::vector<T>(m, T(0)));  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::ones(int n, int m) {  
 auto result = Matrix<T>::zeros(n,m);  
 for (int i = 0, j = 0; i < n && j < m; ++i, ++j) {  
 result[i][j] = T(1);  
 }  
 return result;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::transpose() const {  
 Matrix<T> result = Matrix<T>::zeros(size\_columns(), size\_rows());  
 for (int i = 0; i < size\_rows(); ++i) {  
 for (int j = 0; j < size\_columns(); ++j) {  
 result[j][i] = data[i][j];  
 }  
 }  
 return result;  
}  
  
template <typename T>  
size\_t Matrix<T>::size\_rows() const {  
 return data.size();  
}  
template <typename T>  
auto Matrix<T>::begin() {  
 return data.begin();  
}  
  
template <typename T>  
auto Matrix<T>::begin() const {  
 return data.begin();  
}  
  
template <typename T>  
auto Matrix<T>::end() {  
 return data.end();  
}  
  
template <typename T>  
auto Matrix<T>::end() const {  
 return data.end();  
}  
  
template<typename T>  
Matrix<T>::operator Column<T>() const {  
 if (size\_columns() != 1) {  
 throw std::logic\_error("Wrong cast to column");  
 }  
 Column<T> result;  
 for (int i = 0; i < size\_rows(); ++i) {  
 result.push\_back(data[i][0]);  
 }  
 return result;  
}  
  
template <typename T>  
size\_t Matrix<T>::size\_columns() const {  
 return (data.size() ? data[0].size() : 0);  
}  
  
template <typename T>  
std::vector<T>& Matrix<T>::operator[](size\_t index) {  
 return data[index];  
}  
  
template <typename T>  
const std::vector<T>& Matrix<T>::operator[](size\_t index) const {  
 return data[index];  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::operator \* (T other) const {  
 auto res = \*this;  
 for (int i = 0; i < res.size\_rows(); ++i) {  
 for (int j = 0; j < res.size\_columns(); ++j) {  
 res[i][j] \*= other;  
 }  
 }  
 return res;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::operator \* (const Matrix<T>& other) const {  
 if (size\_columns() != other.size\_rows()) {  
 throw std::logic\_error("cant multiply matrices");  
 }  
 auto result = zeros(size\_rows(), other.size\_columns());  
 for (int i = 0; i < size\_rows(); ++i) {  
 for (int j = 0; j < other.size\_columns(); ++j) {  
 T val = 0;  
 for (int k = 0; k < size\_columns(); ++k) {  
 val += data[i][k] \* other[k][j];  
 }  
 result.data[i][j] = val;  
 }  
 }  
 return result;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::operator + (const Matrix<T>& other) const {  
 if (size\_columns() != other.size\_columns() || size\_rows() != other.size\_rows()) {  
 throw std::logic\_error("cant add matrices");  
 }  
 auto result = zeros(size\_rows(), size\_columns());  
 for (int i = 0; i < size\_rows(); ++i) {  
 for (int j = 0; j < other.size\_columns(); ++j) {  
 result[i][j] = \*this[i][j] + other[i][j];  
 }  
 }  
 return result;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::operator - (const Matrix<T>& other) const {  
 if (size\_columns() != other.size\_columns() || size\_rows() != other.size\_rows()) {  
 throw std::logic\_error("cant subtract matrices");  
 }  
 auto result = zeros(size\_rows(), size\_columns());  
 for (int i = 0; i < size\_rows(); ++i) {  
 for (int j = 0; j < other.size\_columns(); ++j) {  
 result[i][j] = (\*this)[i][j] - other[i][j];  
 }  
 }  
 return result;  
}  
  
template <typename T>  
void Matrix<T>::swap\_rows(int f, int s) {  
 data[f].swap(data[s]);  
}  
  
template <typename T>  
void Matrix<T>::swap\_columns(int f, int s) {  
 for (int i = 0; i < size\_rows(); ++i) {  
 std::swap(data[i][f], data[i][s]);  
 }  
}  
  
template<typename T>  
Matrix<T> to\_column\_matrix(const Column<T>& c) {  
 Matrix<T> result = Matrix<T>::zeros(c.size(), 1);  
 for (int i = 0; i < c.size(); ++i) {  
 result[i][0] = c[i];  
 }  
 return result;  
}  
  
template<typename T>  
Matrix<T> to\_row\_matrix(const Column<T>& c) {  
 return Matrix<T>({c});  
}  
  
  
template <typename T>  
Matrix<T>& Matrix<T>::operator = (const Matrix<T>&other) {  
 data = other.data;  
 return \*this;  
}  
  
template <typename T>  
Matrix<T>& Matrix<T>::operator = (Matrix<T>&& other) {  
 data = std::move(other.data);  
 return \*this;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::concatenate\_columns(const std::vector<Column<T>>& cols) {  
 Matrix<T> result = zeros((cols.size() ? cols[0].size() : 0), cols.size());  
 for (int j = 0; j < cols.size(); ++j) {  
 for (int i = 0; i < cols[j].size(); ++i) {  
 result[i][j] = cols[j][i];  
 }  
 }  
 return result;  
}  
  
template<typename T>  
Column<T> operator + (const Column<T>& lhs, const Column<T>& rhs) {  
 if (lhs.size() != rhs.size()) {  
 throw std::logic\_error("error while summing vectors : sizes are not equal");  
 }  
 Column<T> result(lhs.size());  
 for (int i = 0; i < lhs.size(); ++i) {  
 result[i] = lhs[i] + rhs[i];  
 }  
 return result;  
}  
  
template<typename T>  
Column<T> operator - (const Column<T>& lhs, const Column<T>& rhs) {  
 if (lhs.size() != rhs.size()) {  
 throw std::logic\_error("error while summing vectors : sizes are not equal");  
 }  
 Column<T> result(lhs.size());  
 for (int i = 0; i < lhs.size(); ++i) {  
 result[i] = lhs[i] - rhs[i];  
 }  
 return result;  
}  
  
  
template <typename T>  
Matrix<T> Matrix<T>::construct\_row\_matrix(std::vector<T> v) {  
 return Matrix<T>{{std::move(v)}};  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::construct\_column\_matrix(const std::vector<T>& v) {  
 Matrix<T> result = zeros( v.size(), 1);  
 for (int i = 0; i < v.size(); ++i) {  
 result[i][0] = v[i];  
 }  
 return result;  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::operator \* (const Column<T>& other) const {  
 return \*this \* construct\_column\_matrix(other);  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::construct\_column\_matrix(size\_t size) {  
 return zeros(size,1);  
}  
  
template <typename T>  
Matrix<T> Matrix<T>::construct\_row\_matrix(size\_t size) {  
 return zeros(1,size);  
}  
  
template <typename T>  
Column<T> Matrix<T>::get\_column(size\_t index) const {  
 Column<T> result;  
 for (int i = 0; i < size\_rows(); ++i) {  
 result.push\_back(data[i][index]);  
 }  
 return result;  
}  
  
namespace Math{  
 template <typename T>  
 LUDecomposition<T> lu\_decomposition(const Matrix<T>& other) {  
 Matrix<T> L = Matrix<T>::zeros(other.size\_rows(), other.size\_columns());  
 Matrix<T> U = other;  
 Matrix<T> P = Matrix<T>::ones(other.size\_rows(), other.size\_columns());  
 bool odd = false;  
 int n = other.size\_rows();  
 for (int i = 0; i < n; ++i) {  
 int index = i;  
 for (int j = i; j < n; ++j) {  
 if (std::abs(U[j][i]) > std::abs(U[index][i])) {  
 index = j;  
 }  
 }  
 if (i != index) {  
 L.swap\_rows(i, index);  
 U.swap\_rows(i, index);  
 P.swap\_rows(i, index);  
 odd = !odd;  
 }  
  
  
 L[i][i] = 1;  
 for (int j = i + 1; j < n; ++j) {  
 L[j][i] = U[j][i] / U[i][i];  
 U[j][i] = 0;  
 }  
 for (int j = i + 1; j < n; ++j) {  
 for (int k = i + 1; k < n; ++k) {  
 U[j][k] = U[j][k] - U[i][k] \* L[j][i];  
 }  
 }  
 }  
 return {L, U, P, odd};  
 }  
  
  
  
 template <typename T>  
 Column<T> solve\_linear\_system(const LUDecomposition<T>& dec, Column<T> b) {  
 const auto& [L,U,P,odd] = dec;  
 int n = L.size\_rows();  
 b = Column<T>(P \* b);  
 *// PA = LU  
 //Solve Lz = b* Column<T> z(n);  
 z[0] = b[0];  
 for (int i = 1; i < n; ++i) {  
 T sum = 0;  
 for (int j = 0; j < i; ++j) {  
 sum += L[i][j] \* z[j];  
 }  
 z[i] = b[i] - sum;  
  
 }  
  
 *//Solve Ux = z* Column<T> x(n);  
 x[n - 1] = z[n - 1] / U[n - 1][n - 1];  
 for (int i = n - 2; i >= 0; --i) {  
 T sum = 0;  
 for (int j = i + 1; j < n; ++j) {  
 sum += U[i][j] \* x[j];  
 }  
 x[i] = (z[i] - sum) / U[i][i];  
 }  
  
 *//Return x* return x;  
 }  
  
 template <typename T>  
 Matrix<T> solve\_matrix\_equation(const LUDecomposition<T>& dec, const Matrix<T>& b) {  
 std::vector<Column<T>> result;  
 for (int i = 0; i < dec.L.size\_columns(); ++i) {  
 result.push\_back(solve\_linear\_system(dec, b.get\_column(i)));  
 }  
 return Matrix<T>::concatenate\_columns(result);  
  
 }  
  
 template <typename T>  
 Matrix<T> solve\_matrix\_equation(const Matrix<T>& a, const Matrix<T>& b) {  
 return solve\_matrix\_equation(lu\_decomposition(a), b);  
  
 }  
  
 template <typename T>  
 T determinant(const LUDecomposition<T>& dec) {  
 const auto& [L, U, P, odd] = dec;  
 double Ldet = 1, Udet = 1;  
 int n = L.size\_rows();  
 for (int i = 0; i < n; ++i) {  
 Ldet \*= L[i][i];  
 Udet \*= U[i][i];  
 }  
 return Ldet \* Udet \* (odd ? -1 : 1);  
 }  
  
 template <typename T>  
 long double norm(const Column<T>& d, const std::function<bool(int)>& filter = [] (int) {return true;}) {  
 long double sum = 0;  
 for (int i = 0; i < d.size(); ++i) {  
 if (filter(i)) {  
 sum += d[i] \* d[i];  
 }  
 }  
 return std::sqrt(sum);  
 }  
  
 template <typename T>  
 long double norm(const Matrix<T>& d, const std::function<bool(int,int)>& filter = [] (int,int) {return true;}) {  
 long double sum = 0;  
 for (int i = 0; i < d.size\_rows(); ++i) {  
 for (int j = 0; j < d.size\_columns(); ++j) {  
 if (filter(i, j)) {  
 sum += d[i][j] \* d[i][j];  
 }  
 }  
 }  
 return std::sqrt(sum);  
 }  
  
  
  
 template <typename T>  
 Column<T> sweep\_method(const Matrix<T>& matrix, const Column<T>& d) {  
 struct index\_pair {int i; int j;};  
 std::vector<double> P(d.size()), Q(d.size());  
 int n = d.size();  
 auto M = [&matrix] (index\_pair elem, int inc) -> T {  
 return matrix[elem.i + inc][elem.j + inc];  
 };  
 index\_pair a = {0,-1}, b = {0,0}, c = {0,1};  
 P[0] = -M(c,0) / M(b,0);  
 Q[0] = d[0] / M(b,0);  
 for (int i = 1; i < n - 1; ++i) {  
 P[i] = -M(c,i) / (M(b,i) + M(a,i) \* P[i - 1]);  
 Q[i] = (d[i] - M(a,i) \* Q[i - 1]) / (M(b,i) + M(a,i) \* P[i - 1]);  
 }  
 P[n - 1] = 0;  
 Q[n - 1] = (d[n - 1] - M(a, n - 1) \* Q[n - 2]) / (M(b, n - 1) + M(a, n - 1) \* P[n - 2]);  
 Column<T> x(d.size());  
 x[n - 1] = Q.back();  
 for (int i = x.size() - 2; i >= 0; --i) {  
 x[i] = P[i] \* x[i + 1] + Q[i];  
 }  
 return x;  
 }  
  
 template <typename T>  
 std::vector<Column<T>> simple\_iterations(const Matrix<T>& a, const Column<T>& b, long double epsilon, int max\_iteration\_count = 1'000'000) {  
 int n = b.size();  
 Column<T> beta(n);  
 Matrix<T> alpha = Matrix<T>::zeros(n,n);  
 for (int i = 0; i < n; ++i) {  
 beta[i] = b[i] / a[i][i];  
 }  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j) {  
 alpha[i][j] = -a[i][j] / a[i][i];  
 if (i == j) {  
 alpha[i][j] = 0;  
 }  
 }  
 }  
 std::vector<Column<T>> result = {beta};  
 Column<T> x = beta;  
 for (int i = 0; i < max\_iteration\_count; ++i) {  
 Column<T> new\_x = beta + Column<T>(alpha \* x);  
 double iter\_epsilon = norm(new\_x - x) \* norm(alpha) / (1 - norm(alpha));  
 result.push\_back(new\_x);*//std::cout << "Simple iteration " << i + 1 << " eps " << iter\_epsilon << "\n";* x = new\_x;  
 if (iter\_epsilon <= epsilon) {  
 break;  
 }  
 }  
 return result;  
 }  
  
 template <typename T>  
 std::vector<Column<T>> zeidel(const Matrix<T>& a, const Column<T>& b, long double epsilon, int max\_iteration\_count = 1'000'000) {  
 int n = b.size();  
 Column<T> beta(n);  
 Matrix<T> alpha = Matrix<T>::zeros(n,n);  
 for (int i = 0; i < n; ++i) {  
 beta[i] = b[i] / a[i][i];  
 }  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j) {  
 alpha[i][j] = -a[i][j] / a[i][i];  
 if (i == j) {  
 alpha[i][j] = 0;  
 }  
 }  
 }  
 std::vector<Column<T>> result = {beta};  
 Column<T> x = beta;  
 for (int i = 0; i < max\_iteration\_count; ++i) {  
 Column<T> new\_x(n);  
 for (int j = 0; j < n; ++j) {  
 new\_x[j] = beta[j];  
 for (int k = 0; k < n; ++k) {  
 new\_x[j] += alpha[j][k] \* (k < j ? new\_x[k] : x[k]);  
 }  
 }  
 double iter\_epsilon = (norm(alpha, [] (int i, int j) {return j >= i;}))/(1 - norm(alpha)) \* norm(new\_x - x);  
 result.push\_back(new\_x);*//std::cout << "Zeidel iteration " << i + 1 << " eps " << iter\_epsilon << "\n";* x = new\_x;  
 if (iter\_epsilon <= epsilon) {  
 break;  
 }  
 }  
 return result;  
 }  
  
 template<typename T>  
 struct Eigen {  
 Column<T> eigenvalues;  
 std::vector<Column<T>> eigenvectors;  
 std::vector<long double> epsilons;  
 };  
  
 template <typename T>  
 Eigen<T> rotation\_method(const Matrix<T>& m, long double epsilon, int max\_iteration\_count = 1'000'000) {  
 int n = m.size\_columns();  
 Matrix<T> A = m;  
 Matrix<T> U = Matrix<T>::ones(n,n);  
 std::vector<long double> epsilons;  
 for (int iter = 0; iter < max\_iteration\_count; ++iter) {  
 T max = std::abs(A[0][1]);  
 int max\_i = 0;  
 int max\_j = 1;  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j) {  
 if (i != j && std::abs(A[i][j]) > max) {  
 max = std::abs(A[i][j]);  
 max\_i = i, max\_j = j;  
 }  
 }  
 }  
 Matrix<T> rotate\_matrix = Matrix<T>::ones(n,n);  
 long double phi = (A[max\_i][max\_i] == A[max\_j][max\_j] ?  
 **M\_PI\_4** :  
 atan2(2 \* A[max\_i][max\_j], (A[max\_i][max\_i] - A[max\_j][max\_j])) / 2.);  
 rotate\_matrix[max\_i][max\_i] = rotate\_matrix[max\_j][max\_j] = std::cos(phi);  
 rotate\_matrix[max\_i][max\_j] = -std::sin(phi);  
 rotate\_matrix[max\_j][max\_i] = std::sin(phi);  
 U = U \* rotate\_matrix;  
 A = rotate\_matrix.transpose() \* A \* rotate\_matrix;  
 epsilons.push\_back(norm(A, [] (int i, int j) {return i != j;}));  
 if (epsilons.back() < epsilon) {  
 break;  
 }  
 }  
 Column<T> lambdas(n);  
 for (int i = 0; i < n; ++i) {  
 lambdas[i] = A[i][i];  
 }  
 return {lambdas, U.transpose().get\_data(), epsilons};  
 }  
  
 template <typename T>  
 struct QRDecomposition {  
 Matrix<T> Q;  
 Matrix<T> R;  
 };  
  
 template <typename T>  
 int sign(T v) {  
 return (v < 0 ? 1 : -1);  
 }  
  
 template <typename T>  
 std::pair<std::complex<T>,std::complex<T>> find\_roots(T a0, T a1, T a2, T a3) {  
 T a = 1;  
 T b = - (a0 + a3);  
 T c = a0 \* a3 - a1 \* a2;  
 T discriminant = std::pow(b, 2) - 4 \* a \* c;  
 if (discriminant < 0) {  
 return {{-b / 2., std::sqrt(std::abs(discriminant)) / 2.}, {-b / 2., - std::sqrt(std::abs(discriminant)) / 2.}};  
 }  
 return {{(-b + std::sqrt(discriminant)) / 2.,0},{(-b - std::sqrt(discriminant)) / 2.,0}};  
 }  
  
 template <typename T>  
 QRDecomposition<T> qr\_decomposition(Matrix<T> A) {  
 if (A.size\_columns() != A.size\_rows()) {  
 throw std::logic\_error("Not square matrix");  
 }  
 int n = A.size\_columns();  
 Matrix<T> H = Matrix<T>::ones(n,n);  
  
  
 for (int i = 0; i < n - 1; ++i) {  
 Column<T> a = A.get\_column(i);  
 Column<T> v = a;  
 for (int j = 0; j < i; ++j) {  
 v[j] = 0;  
 }  
 v[i] = a[i] + sign(a[i]) \* norm(a, [i] (int index) {return index >= i;});  
 Matrix<T> v\_col = Matrix<T>::construct\_column\_matrix(v);  
 Matrix<T> v\_row = Matrix<T>::construct\_row\_matrix(v);  
 Matrix<T> H\_cur = Matrix<T>::ones(n,n) - ((v\_col \* v\_row) \* (2 / T(v\_row \* v\_col)));  
  
  
 H = H \* H\_cur;  
 A = H\_cur \* A;  
 }  
 return {H, A};  
 }  
  
 template <typename T>  
 std::vector<T> reverse\_iter\_process(const Matrix<T>& A, T value, long double epsilon, int max\_iteration\_count = 1'000'000) {  
 Column<T> result(A.size\_columns(), 1);  
 Matrix<T> ones = Matrix<T>::ones(A.size\_rows(), A.size\_columns());  
 for (int i = 0; i < max\_iteration\_count; ++i) {  
 Column<T> current\_result = Column<T>(solve\_linear\_system(Math::lu\_decomposition<T>(A - (value \* ones)), result));  
 if (norm(result - (current\_result) / norm(current\_result)) < epsilon) {  
 result = current\_result;  
 break;  
 }  
 result = current\_result / norm(current\_result);  
 }  
 return result;  
 }  
  
 template<typename T>  
 Eigen<std::complex<T>> reverse\_iterations(const Matrix<T>& A, std::vector<std::complex<T>> eigenvalues, long double epsilon, int max\_iteration\_count = 1'000'000) {  
 int n = A.size\_rows();  
 std::vector<std::vector<T>> real\_vectors, imag\_vectors;  
 for (auto num : eigenvalues) {  
 if (num.real() != 0) {  
 real\_vectors.push\_back(reverse\_iter\_process(A, num.real(), epsilon, max\_iteration\_count));  
 } else {  
 real\_vectors.emplace\_back(A.size\_columns());  
 }  
  
 if (num.imag() != 0) {  
 imag\_vectors.push\_back(reverse\_iter\_process(A, num.imag(), epsilon, max\_iteration\_count));  
 } else {  
 imag\_vectors.emplace\_back(A.size\_columns());  
 }  
 }  
 Eigen<std::complex<T>> result;  
 result.eigenvalues = eigenvalues;  
 result.eigenvectors.resize(eigenvalues.size());  
 for (int i = 0; i < real\_vectors.size(); ++i) {  
 for (int j = 0; j < n; ++j) {  
 result.eigenvectors[i].push\_back(std::complex(real\_vectors[i][j], imag\_vectors[i][j]));  
 }  
  
 }  
 return result;  
 }  
  
  
 template <typename T>  
 std::vector<std::complex<T>> qr\_algorithm(Matrix<T> A, long double epsilon, int max\_iteration\_count = 1'000'000) {  
 if (A.size\_columns() != A.size\_rows()) {  
 throw std::logic\_error("not square matrix in qr algo");  
 }  
 int n = A.size\_columns();  
 std::vector<bool> complex\_convergence(n - 1, false);  
 std::vector<bool> real\_convergence(n - 1, false);  
 std::vector<std::complex<T>> complex\_roots(n - 1);  
 std::vector<std::complex<T>> result(n);  
 for (int i = 0; i < n - 1; ++i) {  
 auto [root1, root2] = find\_roots(A[i][i], A[i][i + 1], A[i + 1][i], A[i + 1][i + 1]);  
 complex\_roots[i] = root1;  
 }  
 for (int iter = 0; iter < max\_iteration\_count; ++iter) {  
 auto [Q,R] = qr\_decomposition(A);  
 A = R \* Q;  
  
 for (int j = 0; j < n - 1; ++j) {  
 real\_convergence[j] = (norm(A.get\_column(j), [j] (int index) {return index > j;}) < epsilon);  
 auto [root1, root2] = find\_roots(A[j][j], A[j][j + 1], A[j + 1][j], A[j + 1][j + 1]);  
 if (root1.imag() != 0 && abs(root1 - complex\_roots[j]) < epsilon) {  
 complex\_convergence[j] = true;  
 }  
 complex\_roots[j] = root1;  
 }  
 bool end = true;  
 for (int j = 0; j < n - 1; ++j) {  
 if (real\_convergence[j]) {  
 result[j] = A[j][j];  
 } else if (complex\_convergence[j]) {  
 std::complex<T> root = complex\_roots[j];  
 result[j] = root;  
 result[j + 1] = {root.real(), - root.imag()};  
 ++j;  
 } else {  
 end = false;  
 break;  
 }  
 }  
 std::cout << "iter\n" << iter << "\n" << real\_convergence << "\n" << complex\_convergence << "\n" << A << "\n";  
 if (!complex\_convergence[n - 2]) {  
 result[n - 1] = A[n - 1][n - 1];  
 }  
 if (end) {  
 break;  
 }  
 }  
 return result;  
 }  
  
};  
  
#endif *//LAB1\_MATRIX\_H*

lab1.cpp  
#include<vector>  
#include<cmath>  
#include<iomanip>  
#include "Matrix.h"  
  
using namespace std;  
  
using LMatrix = Matrix<long double>;  
using LColumn = Column<long double>;  
  
int main() {  
 cout << fixed << setprecision(3);  
 LMatrix m({{7, 8, 4, -6},  
 {5, 9, 1, 1},  
 {-1,6,-2, -6},  
 {2, 9,6, -4}});  
 LColumn b{-126,-42,-115,-67};  
 auto decomposition = Math::lu\_decomposition(m);  
 auto x = Math::solve\_linear\_system(decomposition, b);  
 cout << decomposition.L \* decomposition.U << "\n\n";  
 cout << "Matrix:\n" << m << "\nSolution:" << x << "\n\n";  
 auto inverse\_matrix = Math::solve\_matrix\_equation(decomposition, LMatrix::ones(m.size\_rows(), m.size\_columns()));  
 cout << "Inverse matrix:\n" << inverse\_matrix << "\n";  
 cout << "M \* Inverse\n" << m \* inverse\_matrix << "\n";  
 cout << "Determinant = " << Math::determinant(decomposition) << "\n";  
  
}

Результат:

# Лабораторная работа 1.2

Задание: Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

Условие:

Исходный код:

#include<vector>  
#include<cmath>  
#include<iomanip>  
#include "Matrix.h"  
  
using namespace std;  
  
using MatrixType = long double;  
using LMatrix = Matrix<MatrixType>;  
using LColumn = Column<MatrixType>;  
  
int main() {  
 cout << fixed << setprecision(1);  
 LMatrix m({{-6, 6, 0, 0, 0},  
 {2, 10, -7, 0, 0},  
 {0, -8, 18, 9, 0},  
 {0,0,6,-17,-6},  
 {0,0,0,9,14}});  
 LColumn b{30,-31,108, -114, 124};  
  
 cout << "Matrix:\n " << m << "\nValues:\n" << b << "\n\nSolution:\n" << Math::sweep\_method(m,b) << "\n";  
  
}

Результат:  
Matrix:

-6.0 6.0 0.0 0.0 0.0

2.0 10.0 -7.0 0.0 0.0

0.0 -8.0 18.0 9.0 0.0

0.0 0.0 6.0 -17.0 -6.0

0.0 0.0 0.0 9.0 14.0

Values:

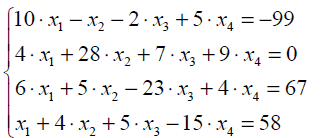
30.0 -31.0 108.0 -114.0 124.0

Solution:

-5.0 0.0 3.0 6.0 5.0

# Лабораторная работа 1.3

Задание: Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Условие:

Исходный код:

#include<vector>  
#include<cmath>  
#include<iomanip>  
#include "Matrix.h"  
  
using namespace std;  
  
  
using MatrixType = long double;  
using LMatrix = Matrix<MatrixType>;  
using LColumn = Column<MatrixType>;  
  
int main() {  
 LMatrix m({{10, -1, -2, 5},  
 {4, 28, 7, 9},  
 {6, 5, -23, 4},  
 {1,4,5,-15}});  
  
 LColumn b{-99,0,67,58};  
 long double epsilon = 0.0001;  
 auto simple\_solution = Math::simple\_iterations(m, b, epsilon);  
 auto zeidel\_solution = Math::zeidel(m, b, epsilon);  
 cout << "Matrix:\n" << m << "\nValues:\n" << b << "\n\n";  
 cout << "Simple iteration method\n";  
 for (int i = 0; i < simple\_solution.size(); ++i) {  
 cout << "iteration: " << i << ", solution" << simple\_solution[i] << "\n";  
 }  
 cout << "\n";  
  
 cout << "Zeidel method\n";  
 for (int i = 0; i < zeidel\_solution.size(); ++i) {  
 cout << "iteration: " << i << ", solution " << zeidel\_solution[i] << "\n";  
 }  
 cout << "\n";  
  
 return 0;  
}

Результат:

Matrix:

10 -1 -2 5

4 28 7 9

6 5 -23 4

1 4 5 -15

Values:

-99 0 67 58

Simple iteration method

iteration: 0, solution-9.9 0 -2.91304 -3.86667

iteration: 1, solution-8.54928 3.3854 -6.16812 -5.49768

iteration: 2, solution-8.04624 4.53047 -5.36345 -5.58988

iteration: 3, solution-7.7247 4.28707 -4.99933 -4.98278

iteration: 4, solution-7.97977 3.95497 -4.86278 -4.90487

iteration: 5, solution-8.02462 3.93223 -4.98797 -4.96492

iteration: 6, solution-8.02191 3.98923 -5.01506 -5.0157

iteration: 7, solution-7.99624 4.01194 -5.01079 -5.00935

iteration: 8, solution-7.99629 4.00516 -4.99805 -5.00016

iteration: 9, solution-7.99901 3.99903 -4.99794 -4.99772

iteration: 10, solution-8.00082 3.99861 -4.99956 -4.9995

iteration: 11, solution-8.0003 3.99985 -5.00043 -5.00028

iteration: 12, solution-7.99996 4.00024 -5.00016 -5.0002

iteration: 13, solution-7.99991 4.0001 -4.99997 -4.99999

iteration: 14, solution-7.99999 3.99998 -4.99995 -4.99996

iteration: 15, solution-8.00001 3.99997 -5 -4.99999

iteration: 16, solution-8.00001 4 -5.00001 -5.00001

iteration: 17, solution-8 4.00001 -5 -5

iteration: 18, solution-8 4 -5 -5

Zeidel method

iteration: 0, solution -9.9 0 -2.91304 -3.86667

iteration: 1, solution -8.54928 3.19244 -5.12174 -5.29255

iteration: 2, solution -7.95883 4.11859 -5.01436 -4.97042

iteration: 3, solution -8.0058 3.99491 -4.99748 -5.0009

iteration: 4, solution -7.99955 3.9996 -5.00013 -5.00012

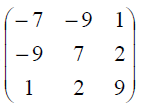
iteration: 5, solution -8.00001 4.00007 -5.00001 -4.99998

iteration: 6, solution -8 4 -5 -5

iteration: 7, solution -8 4 -5 -5

# Лабораторная работа 1.4

Задание: Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Условие:

Исходный код:

#include<vector>  
#include<cmath>  
#include<iomanip>  
#include "Matrix.h"  
  
using namespace std;  
  
  
using MatrixType = long double;  
using LMatrix = Matrix<MatrixType>;  
using LColumn = Column<MatrixType>;  
  
int main() {  
 LMatrix m({{-7,-9,1},  
 {-9,7,2},  
 {1,2,9}});  
  
  
 long double epsilon = 0.0001;  
 auto [eigenvalues, eigenvectors, epsilons] = Math::rotation\_method(m, epsilon);  
 cout << "Eigenvalues\n";  
 cout << eigenvalues << "\n\n";  
 cout << "Eigenvectors\n";  
 for (const auto& v : eigenvectors) {  
 cout << v << "\n";  
 }  
 cout << "\n";  
 for (int i = 0; i < eigenvalues.size(); ++i) {  
 cout << "i: " << i + 1 << ", Ax " << LColumn(m \* eigenvectors[i]) << ", lambda \* x" << eigenvectors[i] \* eigenvalues[i] << "\n";  
 }  
 cout << "\n";  
 for (int i = 0; i < epsilons.size(); ++i) {  
 cout << "epsilon on iteration " << i << ": " << epsilons[i] << "\n";  
 }  
 return 0;  
}

Результат:

Eigenvalues

12.0366 8.51939 -11.556

Eigenvectors

0.368925 -0.827393 -0.423456

0.258828 -0.346119 0.901782

-0.892694 -0.442292 0.0864604

i: 1, Ax 4.4406 -9.95899 -5.09696 , lambda \* x4.4406 -9.95899 -5.09696

i: 2, Ax 2.20505 -2.94872 7.68262 , lambda \* x2.20505 -2.94872 7.68262

i: 3, Ax 10.3159 5.11112 -0.999135 , lambda \* x10.3159 5.11112 -0.999134

epsilon on iteration 0: 3.16228

epsilon on iteration 1: 1.91948

epsilon on iteration 2: 0.165323

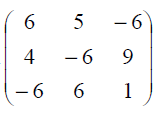
epsilon on iteration 3: 0.0702142

epsilon on iteration 4: 0.000314979

epsilon on iteration 5: 7.78978e-07

# Лабораторная работа 1.5

Задание: Реализовать алгоритм QR-разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений.

Условие:

Исходный код:

#include<vector>  
#include<cmath>  
#include<iomanip>  
#include "Matrix.h"  
  
using namespace std;  
  
  
using MatrixType = long double;  
using LMatrix = Matrix<MatrixType>;  
using LColumn = Column<MatrixType>;  
  
int main() {  
 LMatrix A({{6,5,-6},{4,-6,9},{-6,6,1}});  
 cout << fixed << setprecision(3);  
 auto [Q, R] = Math::qr\_decomposition(A);  
 cout << A << "\n";  
 cout << Q << "\n" << R << "\n";  
 cout << Q \* R << "\n";  
 *//cout << Math::find\_roots(-2.01, -2.58, 0.98, -1.33).first;  
 //cout << "qr\n" << Q \* R << "\n";* vector<complex<long double>> values = Math::qr\_algorithm(A, 0.01);  
 cout << values << "\n";  
 cout << "Test: " << Math::qr\_algorithm(LMatrix({{1,3,5,6},  
 {-1,7,9,2},  
 {-5,4,4,2},  
 {0,1,5,-2}}), 0.01) << "\n";  
 return 0;  
}

Результат:

"D:\Numerical methods\cmake-build-debug\lab5.exe"

6.000 5.000 -6.000

4.000 -6.000 9.000

-6.000 6.000 1.000

Iterations: 29

(re: -13.275, im:0.000) (re: 9.861, im:0.000) (re: 4.415, im:0.000)

# Лабораторная работа 2.1

Задание: Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций

Условие:

Исходный код:

import numpy as np  
import math  
from tabulate import tabulate  
import matplotlib.pyplot as plt  
  
  
def newton(fun, der, x, epsilon=0.001):  
 iter\_info = [[0, x, fun(x)]]  
 iter\_number = 1  
 while True:  
 new\_x = x - fun(x) / der(x)  
 iter\_info.append([iter\_number, new\_x, fun(new\_x)])  
 if np.abs(new\_x - x) < epsilon:  
 return iter\_info  
 x = new\_x  
 iter\_number = iter\_number + 1  
  
def simple\_iteration\_prep(fi\_der, x0, x1, step = 0.001):  
 interval = np.arange(x0, x1, step)  
 return max(np.abs(fi\_der(interval)))  
  
def simple\_iteration(fi, x, q, fun, epsilon = 0.001):  
 iter\_info = [[0, x, fi(x)]]  
 iter\_number = 1  
 while True:  
 new\_x = fi(x)  
 iter\_info.append([iter\_number, new\_x, fun(x)])  
 if (q / (1 - q)) \* np.abs(new\_x - x) < epsilon:  
 return iter\_info  
 x = new\_x  
 iter\_number = iter\_number + 1  
  
  
def secant(fun, x0, x1, epsilon=0.001):  
 iter\_info = [[0, x0, fun(x0)], [1, x1, fun(x1)]]  
 iter\_number = 2  
 while True:  
 new\_x0, new\_x1 = x1, x1 - (fun(x1) \* (x1 - x0))/(fun(x1) - f(x0))  
 iter\_info.append([iter\_number, new\_x1, fun(new\_x1)])  
 if np.abs(new\_x1 - new\_x0) < epsilon:  
 return iter\_info  
 x0, x1 = new\_x0, new\_x1  
 iter\_number = iter\_number + 1  
  
  
def f(x):  
 return np.tan(x) - 5 \* np.power(x,2) + 1  
  
  
*# функция фи, полученная заменой уравнения f(x) = 0 эквивалентным x = fi(x)*def fi(x):  
 return np.sqrt((np.tan(x) + 1) / 5)  
  
  
def fi\_der(x):  
 return 1 / (10 \* np.sqrt((np.tan(x) + 1) / 5) \* np.cos(x)\*\*2)  
  
  
def get\_derivative(fun, epsilon=0.0001):  
 return lambda x: (fun(x + epsilon) - fun(x)) / epsilon  
  
  
t1 = np.arange(-1, 1, 0.0001)  
  
der = get\_derivative(f)  
der2 = get\_derivative(der)  
  
plt.plot(t1, f(t1), label=**"f(x)"**)  
plt.plot(t1, der(t1), label=**"f'(x)"**)  
plt.plot(t1, der2(t1), label=**"f''(x)"**)  
plt.legend()  
plt.grid()  
plt.show()  
  
t2 = np.arange(-np.pi / 4, 1, 0.01)  
  
plt.plot(t2, fi(t2), label=**"phi(x)"**)  
plt.plot(t2, get\_derivative(fi)(t2), label=**"phi'(x)"**)  
*#plt.plot(t2, fi\_der(t2))*plt.xticks(np.arange(-1, 1, 0.1))  
plt.axis([-1, 1, -2, 2])  
plt.legend()  
plt.grid()  
plt.show()  
  
first\_root\_interval = (-0.75, 0)  
second\_root\_interval = (0.5, 0.8)  
  
epsilon = 0.001  
  
newton\_roots = [newton(f, get\_derivative(f, epsilon), -0.5, epsilon), newton(f, get\_derivative(f, epsilon), 0.75, epsilon)]  
secant\_roots = [secant(f, -0.5, -0.5 + 0.01, epsilon), secant(f, 0.75, 0.75 - 0.01, epsilon)]  
  
q = simple\_iteration\_prep(fi\_der, \*second\_root\_interval)  
simple\_iteration\_root = simple\_iteration(fi, -0.3, q, f, epsilon)  
  
print(**"Newton roots"**)  
for info in newton\_roots:  
 print(tabulate(info, headers=[**'iteration'**, **'x'**, **'f(x)'**]))  
  
print(**"Secant roots"**)  
for info in secant\_roots:  
 print(tabulate(info, headers=[**'iteration'**, **'x'**, **'f(x)'**]))  
  
print(**"Simple iterations"**)  
print(**"Chosen interval: [{},{}]"**.format(\*second\_root\_interval))  
print(**"q: {}"**.format(q))  
print(tabulate(simple\_iteration\_root, headers=[**'iteration'**, **'x'**, **'f(x)'**]))

Результат:

Newton roots

iteration x f(x)

----------- --------- ------------

0 -0.5 -0.796302

1 -0.373457 -0.0891958

2 -0.355189 -0.00171645

3 -0.354823 1.25861e-06

iteration x f(x)

----------- -------- -----------

0 0.75 -0.880904

1 0.593684 -0.087397

2 0.574199 -0.00161197

3 0.573826 -2.0867e-06

Secant roots

iteration x f(x)

----------- --------- ------------

0 -0.5 -0.796302

1 -0.49 -0.733888

2 -0.372417 -0.0841173

3 -0.357195 -0.0111442

4 -0.35487 -0.000221577

5 -0.354823 -6.06519e-07

iteration x f(x)

----------- -------- ------------

0 0.75 -0.880904

1 0.74 -0.82491

2 0.592676 -0.0828874

3 0.57622 -0.0103665

4 0.573867 -0.000179662

5 0.573826 -4.05318e-07

Simple iterations

Chosen interval: [0.5,0.8]

q: 0.323352083897837

iteration x f(x)

----------- --------- ----------

0 -0.3 0.371662

1 0.371662 0.240664

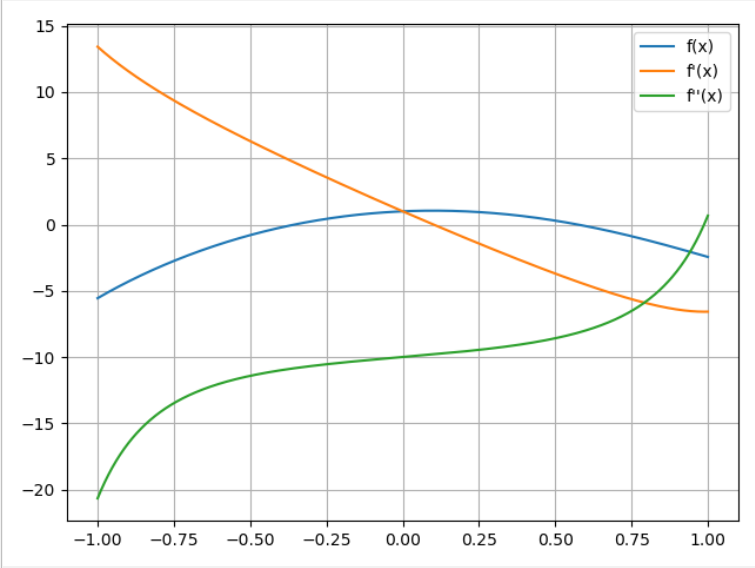
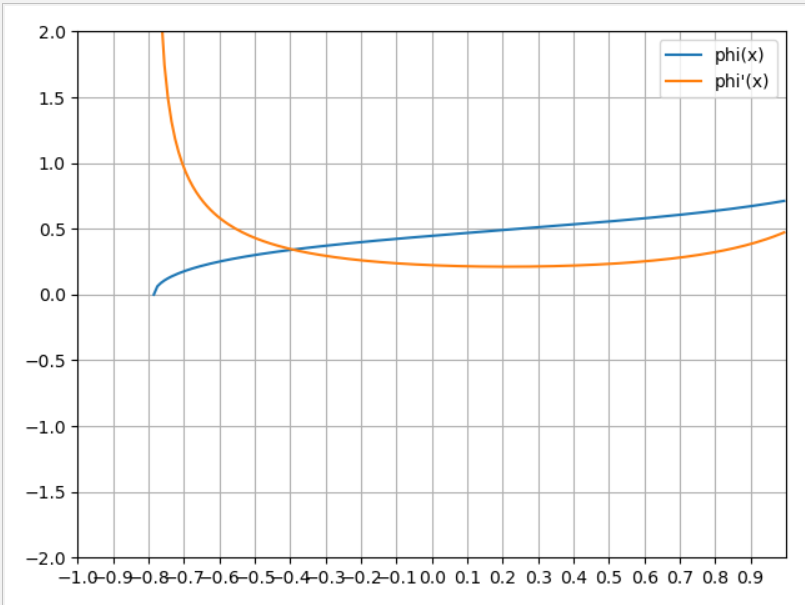
2 0.527215 0.699113

3 0.562527 0.192405

4 0.571047 0.0482934

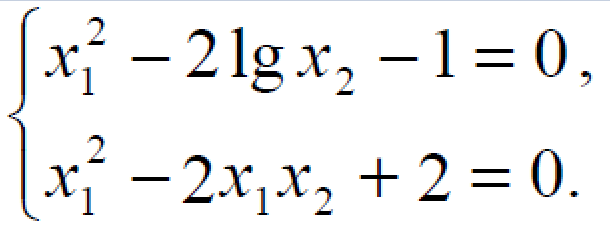
5 0.57314 0.0119721

6 0.573656 0.00296039



# Лабораторная работа 2.2

Задание: Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

Условие:

Исходный код:

import numpy as np  
import math  
import matplotlib.pyplot as plt  
  
from scipy.linalg import lu  
  
def lu\_decomposition(other):  
 n, m = other.shape  
 L = np.zeros(other.shape)  
 U = np.copy(other)  
 P = np.eye(n)  
 odd = False  
 for i in range(0,n):  
 index = i  
 for j in range(i,n):  
 if np.abs(U[j, i]) > np.abs(U[index, i]):  
 index = j  
  
 if i != index:  
 L[[i, index]] = L[[index, i]]  
 U[[i, index]] = U[[index, i]]  
 P[[i, index]] = P[[index, i]]  
 odd = not odd  
  
 L[i, i] = 1  
 for j in range(i + 1, n):  
 L[j][i] = U[j][i] / U[i][i]  
 U[j][i] = 0  
   
 for j in range(i + 1, n):  
 for k in range(i + 1, n):  
 U[j][k] = U[j][k] - U[i][k] \* L[j][i]  
  
 return L, U, P, odd  
  
def solve\_eq(A, b):  
 L,U,P,odd = lu\_decomposition(A)  
 n, m = A.shape  
 b = P.dot(b)  
 z = np.empty(n, dtype=float)  
 z[0] = b[0]  
 for i in range(1, n):  
 sum = 0  
 for j in range(0, i):  
 sum += L[i,j] \* z[j]  
 z[i] = b[i] - sum  
  
 x = np.empty(n, dtype=float)  
 x[n - 1] = z[n - 1] / U[n - 1, n - 1]  
 for i in range(n - 1, -1, -1):  
 sum = 0  
 for j in range(i + 1, n):  
 sum += U[i, j] \* x[j]  
 x[i] = (z[i] - sum) / U[i,i]  
  
 return x  
  
  
def f1(x):  
 return np.power(x[0], 2) - 2 \* np.log10(x[1]) - 1  
  
  
def f2(x):  
 return np.power(x[0], 2) - 2 \* x[0] \* x[1] + 2  
  
  
*# def df1x1(x):  
# return 2 \* x[0]  
#  
#  
# def df1x2(x):  
# return 2/(x[1] \* np.log(10))  
#  
#  
# def df2x1(x):  
# return 2\*x[0] - 2\*x[1]  
#  
#  
# def df2x2(x):  
# return -2\*x[0]*def construct\_equiv\_function(f, var, lbd, coef):  
 return lambda x: (coef \* x[var] - lbd \* f(x)) / coef  
  
def f1\_equiv(x):  
 return np.sqrt(2\*np.log10(x[1]) + 1)  
  
def f2\_equiv(x):  
 return (np.power(x[0], 2) + 2)/(2 \* x[0])  
  
def df1edx1(x):  
 return 0  
  
def df1edx2(x):  
 return 1/(np.sqrt(2 \* np.log10(x[1]) + 1) \* x[1] \* np.log(10))  
  
def df2edx1(x):  
 return 0.5 - 1/(np.power(x[0], 2))  
  
def df2edx2(x):  
 return 0  
  
*# def df1edx1(x):  
# return df1x1(x) + 1  
#  
# def df1edx2(x):  
# return df1x2(x)  
#  
# def df2edx1(x):  
# return df2x1(x)  
#  
# def df2edx2(x):  
# return df2x2(x) + 1*def derivative(fun, var\_num, epsilon=0.001):  
 def res(x):  
 eps\_vector = [(0 if i != var\_num else epsilon) for i in range(0, len(x))]  
 return (fun(x + eps\_vector) - fun(x)) / epsilon  
 return res  
  
def simple\_iterations(functions, x, q, epsilon=0.001):  
 x = np.array(x, dtype=float)  
 while True:  
 new\_x = np.array([f(x) for f in functions], dtype=float)  
 if (q / (1 - q)) \* np.linalg.norm(new\_x - x) < epsilon:  
 return new\_x  
 x = new\_x  
  
def newton(functions, x, epsilon=0.001):  
 jacobi\_matrix = [[derivative(functions[i], j, epsilon) for j in range(0, len(x))] for i in range(0, len(functions))]  
 while True:  
 der\_values = [[der(x) for der in ders] for ders in jacobi\_matrix]  
 b = [-f(x) for f in functions]  
 *# new\_x = x + np.linalg.solve(np.array(der\_values, dtype=float), np.array(b, dtype=float))* dx = solve\_eq(np.array(der\_values, dtype=float), np.array(b, dtype=float))  
 new\_x = x + dx  
 if (np.linalg.norm(x - new\_x) < epsilon):  
 return new\_x  
 x = new\_x  
  
  
def make\_mesh(interval):  
 indices = np.zeros((len(interval),), dtype=np.int64)  
 dims = [len(part) for part in interval]  
 result\_points = []  
 for k in range(0, np.int64(np.prod([len(x) for x in interval]))):  
 result\_points.append([interval[i, indices[i]] for i in range(0, len(indices))])  
 for i in range(0, len(indices)):  
 indices[i] += 1  
 if indices[i] == dims[i]:  
 indices[i] = 0  
 else:  
 break  
 return result\_points  
  
  
  
def test\_function(f, nvar, start, end, step=0.01, ders=None):  
 if ders is None:  
 ders = [derivative(f, i) for i in range(0, nvar)]  
  
 interval = np.array([np.arange(x0, x1, step) for x0, x1 in zip(start, end)])  
 mesh = make\_mesh(interval)  
 a = [[np.abs(der(x)) for x in mesh] for der in ders]  
 b = np.sum(a, axis=0)  
 return max(b)  
  
  
start\_x = np.array([1.1, 1.4])  
  
g\_wide = 0.2  
x1\_int = [1.14 - g\_wide / 2, 1.14 + g\_wide / 2]  
x2\_int = [1.4 - g\_wide / 2, 1.4 + g\_wide / 2]  
  
  
q = max([test\_function(f1\_equiv, 2, x1\_int, x2\_int, 0.001, [df1edx1,df1edx2]),  
 test\_function(f2\_equiv, 2, x1\_int, x2\_int, 0.001, [df2edx1,df2edx2])])  
print(**"Q in simple iteration method: "**, q)  
  
  
epsilon = 0.001  
  
n\_root = newton([f1, f2], start\_x, epsilon)  
si\_root = simple\_iterations([f1\_equiv, f2\_equiv], start\_x, q, epsilon)  
print(n\_root, f1(n\_root), f2(n\_root))  
print(si\_root, f1(si\_root), f2(si\_root))

Результаты:

Q in simple iteration method: 0.4245562130177517

[1.14876666 1.44488213] 1.0227572122545325e-09 1.2587391129414982e-09

[1.14851978 1.44466795] -0.0004383808306007708 0.0006382512444782495

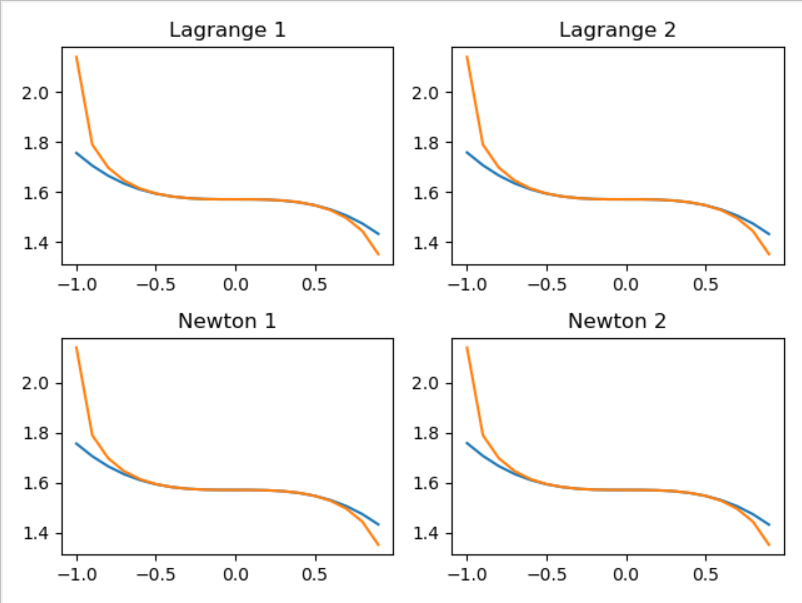
# Лабораторная работа 3.1

Задание: Используя таблицу значений функции , вычисленных в точках , построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке

Условие:

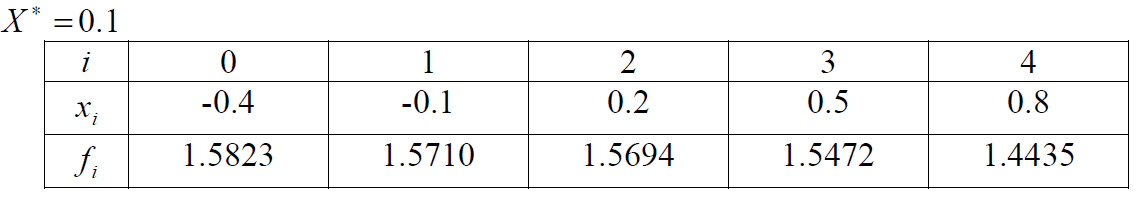
Исходный код:

import numpy as np  
import math  
import matplotlib.pyplot as plt  
  
  
def f(x):  
 return np.arccos(x) + x  
  
  
points1 = np.array([-0.4, -0.1, 0.2, 0.5])  
points2 = np.array([-0.4, 0, 0.2, 0.5])  
  
  
def Lagrange(points, values=None, function=None):  
 if values is None and function is None:  
 raise ValueError(**"function needs function values or function itself"**)  
  
 if values is None:  
 values = function(points)  
  
 if len(values) != len(points):  
 raise ValueError(**"length is wrong"**)  
  
 values = values[:]  
 points = points[:]  
 n = len(values)  
 return lambda x: np.sum(  
 [values[i] \* np.prod([1 if j == i else ((x - points[j]) / (points[i] - points[j])) for j in range(0, n)]) for i  
 in range(0, n)], axis = 0)  
  
def divided\_diff(points, function):  
 if (len(points) == 1):  
 return function(points[0])  
  
 if (len(points) == 2):  
 return (function(points[0]) - function(points[1])) / (points[0] - points[1])  
  
 n = len(points)  
 return (divided\_diff(points[:n-1], function) - divided\_diff(points[1:], function)) / (points[0] - points[n - 1])  
  
def Newton(points, function):  
 points = points[:]  
 n = len(points)  
 cur\_arr = []  
 divided\_sums\_counted = []  
 for i in range(0, n):  
 cur\_arr.append(points[i])  
 divided\_sums\_counted.append(divided\_diff(cur\_arr, function))  
  
 def result(x):  
 cur\_prod = 1  
 res\_sum = 0  
 for i in range(0, n):  
 res\_sum += divided\_sums\_counted[i] \* cur\_prod  
 cur\_prod \*= (x - points[i])  
 return res\_sum  
  
 return result  
  
  
  
interval = np.arange(-1, 1, 0.1)  
  
  
plt.figure()  
plt.subplot(221)  
plt.plot(interval, [Lagrange(points1, function=f)(x) for x in interval])  
plt.plot(interval, f(interval))  
plt.title(**"Lagrange 1"**)  
  
plt.subplot(222)  
plt.plot(interval, [Lagrange(points2, function=f)(x) for x in interval])  
plt.plot(interval, f(interval))  
plt.title(**"Lagrange 2"**)  
  
plt.subplot(223)  
plt.plot(interval, [Newton(points1, function=f)(x) for x in interval])  
plt.plot(interval, f(interval))  
plt.title(**"Newton 1"**)  
  
plt.subplot(224)  
plt.plot(interval, [Newton(points2, function=f)(x) for x in interval])  
plt.plot(interval, f(interval))  
plt.title(**"Newton 2"**)  
plt.show()

Результат:

# Лабораторная работа 3.2

Задание: Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при и . Вычислить значение функции в точке

Условие: 

Исходный код:

import numpy as np  
import math  
import matplotlib.pyplot as plt  
  
  
def sweep\_method(matrix, d):  
 P = np.zeros((len(d),))  
 Q = np.zeros((len(d),))  
 n = len(d)  
 M = lambda i, j, inc: matrix[i+inc, j+inc]  
 a = lambda i: M(0, -1, i)  
 b = lambda i: M(0, 0, i)  
 c = lambda i: M(0, 1, i)  
 P[0] = -c(0) / b(0)  
 Q[0] = d[0] / b(0)  
 for i in range(1, n - 1):  
 P[i] = -c(i) / (b(i) + a(i) \* P[i - 1])  
 Q[i] = (d[i] - a(i) \* Q[i - 1]) / (b(i) + a(i) \* P[i - 1])  
 P[n - 1] = 0  
 Q[n - 1] = (d[n - 1] - a(n - 1) \* Q[n - 2]) / (b(n - 1) + a(n - 1) \* P[n - 2])  
 x = np.zeros(n)  
 x[n - 1] = Q[-1]  
 for i in range(n - 2, -1, -1):  
 x[i] = P[i] \* x[i + 1] + Q[i]  
 return x  
  
  
def make\_spline(a,b,c,d, points):  
 a = np.copy(a)  
 b = np.copy(b)  
 c = np.copy(c)  
 d = np.copy(d)  
 points = np.copy(points)  
  
 def result(x):  
 i = np.searchsorted(points, x)  
 if i == 0 and x == points[0]:  
 i += 1  
 if i == 0 or i == len(points):  
 raise ValueError(**"This point {} doesnt belong to spline"**.format(x))  
 i -= 1  
 return a[i] + b[i] \* (x - points[i]) + c[i] \* np.power((x - points[i]),2) + d[i] \* np.power((x - points[i]), 3)  
  
 return result  
  
def find\_coefs(points, values):  
 *#points, values - n + 1* n = len(points) - 1  
 h = [points[i] - points[i - 1] for i in range(1, n + 1)]  
 A = np.zeros((n-1,n-1), dtype=np.float32)  
  
 r = np.array([3 \* (((values[i + 1] - values[i]) / h[i]) - ((values[i] - values[i - 1]) / h[i - 1])) for i in range(1, n)])  
  
 last = n - 2  
 A[0,0] = 2 \* (h[0] + h[1])  
 A[0,1] = h[1]  
 A[last, last - 1] = h[n - 2]  
 A[last, last] = 2 \* (h[n - 2] + h[n - 1])  
 for i in range(1, last):  
 A[i, i - 1] = h[i]  
 A[i, i] = 2 \* (h[i] + h[i + 1])  
 A[i, i + 1] = h[i]  
  
 *# c = np.linalg.solve(A, r)* c = sweep\_method(A, r)  
 c = np.insert(c,0,0)  
 a = [values[i] for i in range(0, n)]  
 b = [(values[i] - values[i - 1]) / h[i - 1] - (1/3) \* h[i - 1] \* (c[i] + 2 \* c[i - 1]) for i in range(1, n)]  
 b.append((values[-1] - values[-2]) / h[-1] - (2/3) \* h[-1] \* c[-1])  
 d = [(c[i + 1] - c[i]) / (3 \* h[i]) for i in range(0, n - 1)]  
 d.append(-(c[-1] / (3 \* h[-1])))  
 return a,b,c,d  
  
points = [-0.4, -0.1, 0.2, 0.5, 0.8]  
values = [1.5823, 1.5710, 1.5694, 1.5472, 1.4435]  
  
  
a,b,c,d = find\_coefs(points, values)  
for a0, b0, c0, d0 in zip(a,b,c,d):  
 print(**"{}x^3 + {}x^2 + {}x + {}"**.format(a0, b0, c0, d0))  
  
spline\_function = make\_spline(\*find\_coefs(points, values), points)  
  
interval = np.linspace(-0.4, 0.8, 120, endpoint=False)  
plt.plot(points, values, **"ro"**)  
plt.plot(interval, [spline\_function(x) for x in interval])  
plt.show()

Результат: 1.5823x^3 + -0.04638095203467811x^2 + 0.0x + 0.09682539297790173

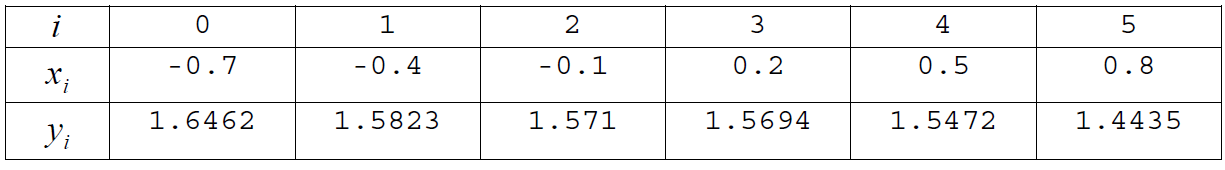
1.571x^3 + -0.020238094645833458x^2 + 0.08714285368011157x + -0.12486771990592774

1.5694x^3 + -0.001666669540935084x^2 + -0.025238094235223413x + -0.7195766909833099

1.5472x^3 + -0.21109524344262584x^2 + -0.6728571161202023x + 0.7476190179113358

# Лабораторная работа 3.3

Задание: Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены a) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов

Условие:

Исходный код:

import numpy as np  
import math  
import matplotlib.pyplot as plt  
  
  
def lu\_decomposition(other):  
 n, m = other.shape  
 L = np.zeros(other.shape)  
 U = np.copy(other)  
 P = np.eye(n)  
 odd = False  
 for i in range(0, n):  
 index = i  
 for j in range(i, n):  
 if np.abs(U[j, i]) > np.abs(U[index, i]):  
 index = j  
  
 if i != index:  
 L[[i, index]] = L[[index, i]]  
 U[[i, index]] = U[[index, i]]  
 P[[i, index]] = P[[index, i]]  
 odd = not odd  
  
 L[i, i] = 1  
 for j in range(i + 1, n):  
 L[j][i] = U[j][i] / U[i][i]  
 U[j][i] = 0  
  
 for j in range(i + 1, n):  
 for k in range(i + 1, n):  
 U[j][k] = U[j][k] - U[i][k] \* L[j][i]  
  
 return L, U, P, odd  
  
  
def solve\_eq(A, b):  
 L, U, P, odd = lu\_decomposition(A)  
 n, m = A.shape  
 b = P.dot(b)  
 z = np.empty(n, dtype=float)  
 z[0] = b[0]  
 for i in range(1, n):  
 sum = 0  
 for j in range(0, i):  
 sum += L[i, j] \* z[j]  
 z[i] = b[i] - sum  
  
 x = np.empty(n, dtype=float)  
 x[n - 1] = z[n - 1] / U[n - 1, n - 1]  
 for i in range(n - 1, -1, -1):  
 sum = 0  
 for j in range(i + 1, n):  
 sum += U[i, j] \* x[j]  
 x[i] = (z[i] - sum) / U[i, i]  
  
 return x  
  
*#метод наименьших квадратов*x = np.array([-0.7, -0.4, -0.1, 0.2, 0.5, 0.8])  
y = np.array([1.6462, 1.5823, 1.571, 1.5694, 1.5472, 1.4435])  
  
*# x = [0.0, 1.7, 3.4, 5.1, 6.8, 8.5]  
# y = [0.4713, 1.0114, 1.5515, 2.0916, 2.6317, 3.1718]*def count\_mse(approx, x, y):  
 return np.sum(np.square(approx(x) - y))  
  
def LSM(x, y, polynom\_power):  
 n = polynom\_power + 1  
 m = len(x)  
 A = np.zeros((n,n), dtype=np.float32)  
  
 A[-1] = np.array([np.sum([np.power(x\_cur,i) for x\_cur in x]) for i in range(n - 1, -1, -1)])  
 for i in range(n - 2, -1, -1):  
 A[i,1:n] = A[i + 1, 0:n-1]  
 A[i, 0] = np.sum([np.power(x\_cur,2\*n - i - 2) for x\_cur in x])  
  
 b = np.array([np.sum([np.power(x[j], i) \* y[j] for j in range(0, m)]) for i in range(0, n)], dtype=np.float32)  
 b = np.flip(b)  
  
 coefs = solve\_eq(A, b)  
  
 return lambda arg: np.sum([coefs[i] \* np.power(arg, len(coefs) - i - 1) for i in range(0, len(coefs))], axis=0)  
  
linear\_approximator = LSM(x,y,1)  
square\_approximator = LSM(x,y,2)  
cubic\_approximator = LSM(x,y,3)  
  
interval = np.arange(-1, 1, 0.01)  
  
  
plt.plot(x, y, **'ro'**, label=**"points"**)  
plt.plot(interval, linear\_approximator(interval), label=**"linear approximator"**)  
plt.plot(interval, square\_approximator(interval), label=**"square approximator"**)  
plt.plot(interval, cubic\_approximator(interval), label=**"cubic approximator"**)  
  
print(**"Mean squared errors:**\n**linear {}**\n**square {}**\n**cubic {}"**.format(count\_mse(linear\_approximator, x, y),  
 count\_mse(square\_approximator, x, y),  
 count\_mse(cubic\_approximator, x, y)))  
  
  
plt.legend()  
plt.show()

Результат:

Mean squared errors:

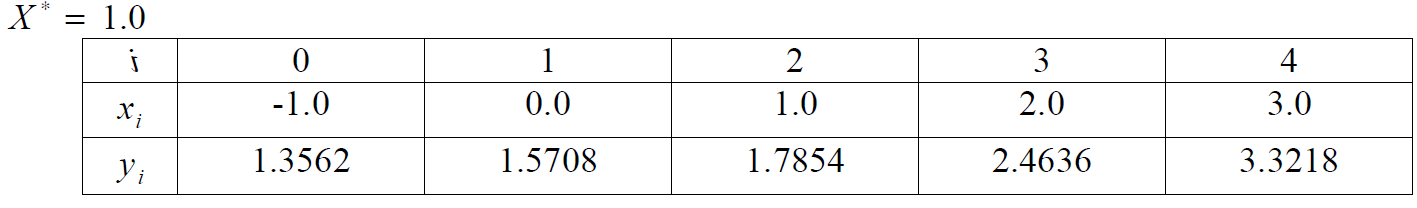
linear 0.003940351047632441

square 0.0032396991428675494

cubic 1.8977142962301985e-05

# Лабораторная работа 3.4

Задание: Вычислить первую и вторую производную от таблично заданной функции в точке

Условие:

Исходный код:

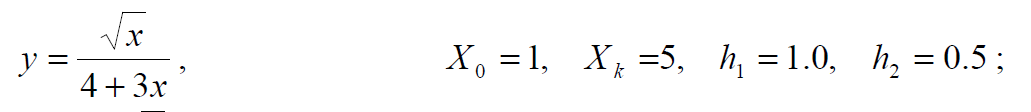
import numpy as np  
import math  
import matplotlib.pyplot as plt  
  
x = np.array([-1.0, 0, 1.0, 2.0, 3.0])  
y = np.array([1.3562, 1.5708, 1.7854, 2.4636, 3.3218])  
  
  
*# x = np.array([0.0, 0.1, 0.2, 0.3, 0.4], dtype=np.float64)  
# y = np.array([1.0, 1.1052, 1.2214, 1.3499, 1.4918], dtype=np.float64)*def get\_polynom(x,y):  
 x = np.copy(x)  
 y = np.copy(y)  
 def polynom(arg):  
 i = np.searchsorted(x, arg)  
 if (arg == x[0]):  
 i = 1  
 i -= 1  
 return y[i] \  
 + (arg - x[i]) \* (y[i + 1] - y[i]) / (x[i + 1] - x[i]) \  
 + (arg - x[i]) \* (arg - x[i + 1]) \* ((y[i + 2] - y[i + 1])/(x[i + 2] - x[i + 1]) - (y[i + 1] - y[i])/(x[i + 1] - x[i])) / (x[i + 2] - x[i])  
 return polynom  
  
  
  
def get\_polynom\_derivative(x,y):  
 x = np.copy(x)  
 y = np.copy(y)  
  
 def polynom(arg):  
 i = np.searchsorted(x, arg)  
 if (arg == x[0]):  
 i = 1  
 i -= 1  
 return (y[i + 1] - y[i])/(x[i + 1] - x[i]) + \  
 ((y[i + 2] - y[i + 1])/(x[i + 2] - x[i + 1]) - (y[i + 1] - y[i])/(x[i + 1] - x[i])) \* (2 \* arg - x[i] - x[i + 1]) / (x[i + 2] - x[i])  
  
 return polynom  
  
def get\_polynom\_derivative2(x,y):  
 x = np.copy(x)  
 y = np.copy(y)  
  
 def polynom(arg):  
 i = np.searchsorted(x, arg)  
 if (arg == x[0]):  
 i = 1  
 i -= 1  
 return 2 \* ((y[i + 2] - y[i + 1])/(x[i + 2] - x[i + 1]) - (y[i + 1] - y[i])/(x[i + 1] - x[i])) / (x[i + 2] - x[i])  
  
 return polynom  
  
def approx\_polynom(x,y):  
 x = np.copy(x)  
 y = np.copy(y)  
 return get\_polynom(x,y), get\_polynom\_derivative(x,y), get\_polynom\_derivative2(x,y)  
  
f, fd, fd2 = approx\_polynom(x,y)  
  
print(**"f(x) = {}, f'(x) = {}, f''(x) = {}"**.format(f(0.2), fd(0.2), fd2(0.2)))

Результат:

f(x) = 1.576632, f'(x) = 0.0755200000000002, f''(x) = 0.4635999999999998

# Лабораторная работа 3.5

Задание: Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

Условие:

Исходный код:

import numpy as np  
import math  
import matplotlib.pyplot as plt  
  
def f(x):  
 return np.sqrt(x) / (4 + 3 \* x)  
  
def rect\_method(x, f):  
 return np.sum([f((x[i + 1] + x[i]) / 2) \* (x[i + 1] - x[i]) for i in range(0,len(x) - 1)])  
  
def trapeze\_method(x, f):  
 y = f(x)  
 return np.sum([(y[i + 1] + y[i]) \* (x[i + 1] - x[i]) for i in range(0, len(x) - 1)]) / 2.  
  
def simpson\_method(x, f):  
 y = f(x)  
 return np.sum([(f(x[i]) + 4 \* f((x[i] + x[i + 1]) / 2) + f(x[i + 1])) \* (x[i + 1] - x[i]) for i in range(0, len(x) - 1)]) / 6.  
  
rect\_method.runge\_coef = 2  
trapeze\_method.runge\_coef = 2  
simpson\_method.runge\_coef = 4  
  
def integrate(method, function, x0, xk, h):  
 res = method(np.arange(x0, xk, h), f)  
 return {**"result"**: res, **"h"**: h}  
  
  
  
def print\_results(name, result\_list, err):  
 print(**"method: {}, results: {}, errors: {}"**.format(name,result\_list, err))  
  
  
*# analitic\_solution = 0.5312191613*def test\_method(method, function, x0, xk, h):  
 result\_list = [integrate(method, function, x0, xk, h\_cur) for h\_cur in h]  
 k = result\_list[0][**"h"**] / result\_list[1][**"h"**]  
 err = np.abs(result\_list[0][**"result"**] - result\_list[1][**"result"**]) / (np.power(k, method.runge\_coef) - 1)  
 *# err\_analitic = np.abs(result\_list[0]["result"] - analitic\_solution) / (np.power(k, method.runge\_coef) - 1)* return result\_list, err  
  
x0 = 1  
xk = 5  
h = [1.0, 0.5, 0.1, 0.01]  
  
print(**"h: {}"**.format(h))  
print\_results(**"rectangle method"**, \*test\_method(rect\_method, f, x0, xk, h))  
print\_results(**"trapeze method"**, \*test\_method(trapeze\_method, f, x0, xk, h))  
print\_results(**"simpson method"**, \*test\_method(simpson\_method, f, x0, xk, h))

Результат:

h: [1.0, 0.5, 0.1, 0.01]

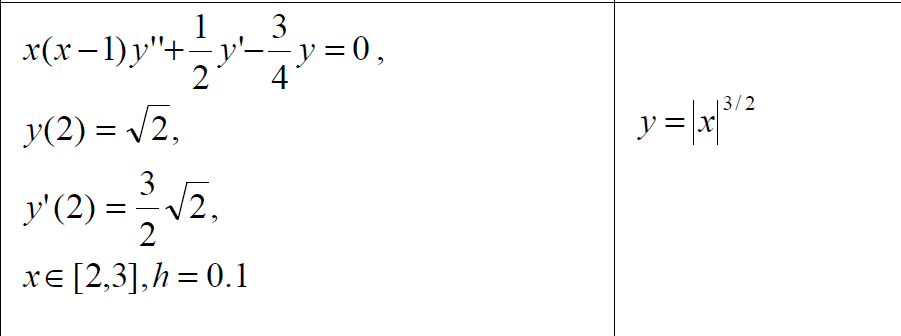
method: rectangle method, results: [{'result': 0.4106006334886676, 'h': 1.0}, {'result': 0.4716775772793693, 'h': 0.5}, {'result': 0.5194232701621695, 'h': 0.1}, {'result': 0.5300420135750599, 'h': 0.01}], errors: 0.02035898126356724

method: trapeze method, results: [{'result': 0.40858460517117917, 'h': 1.0}, {'result': 0.47114719566648966, 'h': 0.5}, {'result': 0.5194019044680592, 'h': 0.1}, {'result': 0.5300418007362198, 'h': 0.01}], errors: 0.02085419683177016

method: simpson method, results: [{'result': 0.40992862404950475, 'h': 1.0}, {'result': 0.47150078340840934, 'h': 0.5}, {'result': 0.519416148264133, 'h': 0.1}, {'result': 0.5300419426287799, 'h': 0.01}], errors: 0.004104810623926972

# Лабораторная работа 4.1

Задание: Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Условие:

Исходный код:

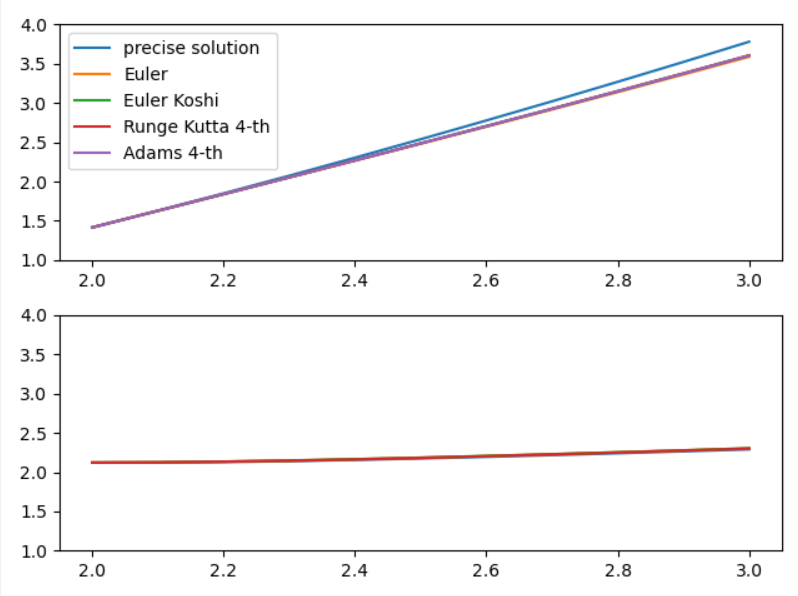
import numpy as np  
import math  
from tabulate import tabulate  
import matplotlib.pyplot as plt  
  
  
def precise\_solution(x):  
 return np.power(np.abs(x), 1.5) - np.sqrt(2)  
  
  
def f1(x, y1, y2):  
 return y2  
  
  
def f2(x, y1, y2):  
 return (0.5 \* y2 - 0.75 \* y1) / (x \* (1 - x))  
  
  
class Euler:  
 def \_\_init\_\_(self, functions, x0, start\_conditions):  
 self.functions = functions  
 self.x0 = x0  
 self.start\_conditions = start\_conditions  
 self.label = **"Euler"** self.last\_points = None  
 self.last\_values = None  
 self.p = 1  
  
 def integrate(self, x\_end, h):  
 points = [self.x0]  
 values = [self.start\_conditions] *# type: list* while points[-1] <= x\_end:  
 cur\_x = points[-1]  
 cur\_condition = values[-1]  
 new\_x = cur\_x + h  
 new\_condition = [cur\_condition[i] + h \* self.functions[i](cur\_x, \*cur\_condition) for i in range(0, len(cur\_condition))]  
 points.append(new\_x)  
 values.append(new\_condition)  
 self.last\_points = np.array(points)  
 self.last\_values = np.array(values)  
 return np.array(points), np.array(values)  
  
class EulerKoshi:  
 def \_\_init\_\_(self, functions, x0, start\_conditions):  
 self.functions = functions  
 self.x0 = x0  
 self.start\_conditions = start\_conditions  
 self.label = **"Euler Koshi"** self.last\_points = None  
 self.last\_values = None  
 self.p = 2  
  
 def integrate(self, x\_end, h):  
 points = [self.x0]  
 values = [self.start\_conditions] *# type: list* while points[-1] <= x\_end:  
 cur\_x = points[-1]  
 cur\_condition = values[-1]  
 new\_x = cur\_x + h  
 new\_condition = [cur\_condition[i] + h \* self.functions[i](cur\_x, \*cur\_condition) for i in range(0, len(cur\_condition))]  
 new\_condition\_corrected = [cur\_condition[i] + h \* (self.functions[i](cur\_x, \*cur\_condition) + self.functions[i](new\_x, \*new\_condition)) / 2 for i in range(0, len(cur\_condition))]  
 points.append(new\_x)  
 values.append(new\_condition\_corrected)  
 self.last\_points = np.array(points)  
 self.last\_values = np.array(values)  
 return np.array(points), np.array(values)  
  
  
class EulerAdvanced:  
 def \_\_init\_\_(self, functions, x0, start\_conditions):  
 self.functions = functions  
 self.x0 = x0  
 self.start\_conditions = start\_conditions  
 self.last\_points = None  
 self.last\_values = None  
 self.label = **"Euler advanced"** self.p = 2  
  
 def integrate(self, x\_end, h):  
 points = [self.x0]  
 values = [self.start\_conditions] *# type: list* while points[-1] <= x\_end:  
 cur\_x = points[-1]  
 cur\_condition = values[-1]  
 new\_x = cur\_x + h  
 half\_condition = [cur\_condition[i] + (h / 2) \* self.functions[i](cur\_x, \*cur\_condition) for i in range(0, len(cur\_condition))]  
 new\_condition = [cur\_condition[i] + h \* self.functions[i](cur\_x + h / 2, \*half\_condition) for i in range(0, len(cur\_condition))]  
 points.append(new\_x)  
 values.append(new\_condition)  
 self.last\_points = np.array(points)  
 self.last\_values = np.array(values)  
 return np.array(points), np.array(values)  
  
  
class RungeKutta:  
 def \_\_init\_\_(self, functions, x0, start\_conditions):  
 self.functions = functions  
 self.x0 = x0  
 self.start\_conditions = start\_conditions  
 self.last\_points = None  
 self.last\_values = None  
 self.label = **"Runge Kutta 4-th"** self.p = 4  
  
 def integrate(self, x\_end, h):  
 points = np.array([self.x0])  
 values = np.array([self.start\_conditions])  
 while points[-1] <= x\_end:  
 cur\_x = points[-1]  
 cur\_condition = values[-1]  
 k1 = np.array([h \* self.functions[i](cur\_x, \*cur\_condition) for i in range(0, len(cur\_condition))])  
 k2 = np.array([h \* self.functions[i](cur\_x + h / 2, \*(cur\_condition + (k1 / 2))) for i in range(0, len(cur\_condition))])  
 k3 = np.array([h \* self.functions[i](cur\_x + h / 2, \*(cur\_condition + (k2 / 2))) for i in range(0, len(cur\_condition))])  
 k4 = np.array([h \* self.functions[i](cur\_x + h, \*(cur\_condition + k3)) for i in range(0, len(cur\_condition))])  
 new\_x = cur\_x + h  
 new\_condition = cur\_condition + (k1 + 2 \* k2 + 2 \* k3 + k4) / 6  
 num = k2 - k3  
 den = k1 - k2  
 *# if np.min(np.abs(den)) == 0 or np.mean(np.abs(num / den)) >= 0.1:  
 # h /= 2  
 # elif np.mean(np.abs(num / den)) < 0.01:  
 # h \*= 2* points = np.append(points, [new\_x], axis=0)  
 values = np.append(values, [new\_condition], axis=0)  
  
 self.last\_points = np.array(points)  
 self.last\_values = np.array(values)  
 return np.array(points), np.array(values)  
  
  
class Adams:  
 def \_\_init\_\_(self, functions, x0, start\_conditions):  
 self.functions = functions  
 self.x0 = x0  
 self.start\_conditions = start\_conditions  
 self.last\_points = None  
 self.last\_values = None  
 self.label = **"Adams 4-th"** self.p = 4  
  
 def integrate(self, x\_end, h):  
 points = np.array(self.x0)  
 values = np.array(self.start\_conditions)  
 while points[-1] <= x\_end:  
 cur\_points = points[-4:]  
 cur\_values = values[-4:]  
 f = np.array([[self.functions[i](cur\_points[j], \*cur\_values[j]) for i in range(0, len(self.functions))] for j in range(0,len(cur\_points))])  
 new\_values = np.array([cur\_values[-1,i] + (h/24)\*(55 \* f[-1,i] - 59 \* f[-2,i] + 37 \* f[-3,i] - 9 \* f[-4,i]) for i in range(0, len(self.functions))])  
 new\_x = points[-1] + h  
 points = np.append(points, [new\_x], axis=0)  
 values = np.append(values, [new\_values], axis=0)  
 self.last\_points = points  
 self.last\_values = values  
 return points, values  
  
fig, ax = plt.subplots(2)  
  
ax[0].set\_ylim([1, 4])  
ax[1].set\_ylim([1, 4])  
  
def plot\_solved(solver):  
 for i in range(0, solver.last\_values.shape[-1]):  
 ax[i].plot(solver.last\_points, solver.last\_values[:, i], label=solver.label)  
  
x0 = 2  
x1 = 3  
start\_conditions = [np.sqrt(2), 1.5 \* np.sqrt(2)]  
h = 0.1  
  
x\_interval = np.arange(x0, x1 + h/2, h)  
ax[0].plot(x\_interval, precise\_solution(x\_interval), label=**"precise solution"**)  
  
solvers = [Euler, EulerKoshi, RungeKutta]  
  
adams\_start\_x = None  
adams\_start\_y = None  
  
solver\_instances = []  
  
for solver\_class in solvers:  
 solver = solver\_class([f1, f2], x0, start\_conditions)  
 points, values = solver.integrate(x1, h)  
 if (solver.\_\_class\_\_.\_\_name\_\_ == **"RungeKutta"**):  
 adams\_start\_x = points[:3]  
 adams\_start\_y = values[:3]  
 solver\_instances.append(solver)  
 plot\_solved(solver)  
  
  
  
x0 = np.append([x0], adams\_start\_x, axis=0)  
start\_conditions = np.append([start\_conditions], adams\_start\_y, axis=0)  
  
adams = Adams([f1,f2], x0, start\_conditions)  
adams.integrate(x1,h)  
  
solver\_instances.append(adams)  
  
plot\_solved(adams)  
  
ax[0].legend()  
plt.show()  
  
for solver in solver\_instances:  
 points1, values1 = solver.integrate(x1, h / 2)  
 points1 = points1[::2]  
 values1 = values1[::2]  
  
 points2, values2 = solver.integrate(x1, h)  
  
 if (values1.shape[0] > values2.shape[0]):  
 values1 = np.resize(values1, values2.shape)  
 else:  
 values2 = np.resize(values2, values1.shape)  
  
 error = np.linalg.norm(np.abs(values1[:, 0] - values2[:, 0]) / (np.power(2, solver.p) - 1))  
 print(**"{} error: {}"**.format(solver.label, error))

Результат:

Euler error: 0.014696709883703106

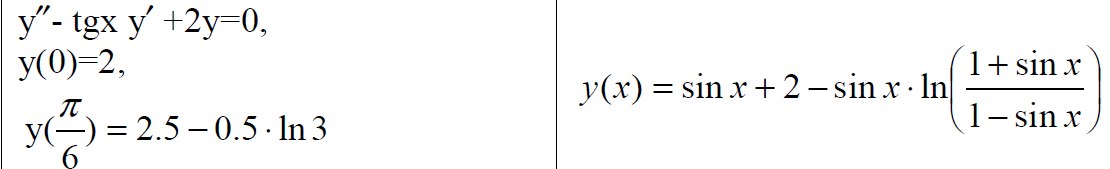
Euler Koshi error: 0.0004363140761390785

Runge Kutta 4-th error: 8.265104990668217e-08

Adams 4-th error: 0.06775977573838743

# Лабораторная работа 4.2

Задание: Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Условие:

Исходный код:

import numpy as np  
import math  
from tabulate import tabulate  
import matplotlib.pyplot as plt  
  
class RungeKutta:  
 def \_\_init\_\_(self, functions, x0, start\_conditions):  
 self.functions = functions  
 self.x0 = x0  
 self.start\_conditions = start\_conditions  
 self.last\_points = None  
 self.last\_values = None  
 self.label = **"Runge Kutta 4-th"** self.p = 4  
  
 def integrate(self, x\_end, h):  
 points = np.array([self.x0])  
 values = np.array([self.start\_conditions])  
 while points[-1] <= x\_end:  
 cur\_x = points[-1]  
 cur\_condition = values[-1]  
 k1 = np.array([h \* self.functions[i](cur\_x, \*cur\_condition) for i in range(0, len(cur\_condition))])  
 k2 = np.array([h \* self.functions[i](cur\_x + h / 2, \*(cur\_condition + (k1 / 2))) for i in range(0, len(cur\_condition))])  
 k3 = np.array([h \* self.functions[i](cur\_x + h / 2, \*(cur\_condition + (k2 / 2))) for i in range(0, len(cur\_condition))])  
 k4 = np.array([h \* self.functions[i](cur\_x + h, \*(cur\_condition + k3)) for i in range(0, len(cur\_condition))])  
 new\_x = cur\_x + h  
 new\_condition = cur\_condition + (k1 + 2 \* k2 + 2 \* k3 + k4) / 6  
 num = k2 - k3  
 den = k1 - k2  
 *# if np.min(np.abs(den)) == 0 or np.mean(np.abs(num / den)) >= 0.1:  
 # h /= 2  
 # elif np.mean(np.abs(num / den)) < 0.01:  
 # h \*= 2* points = np.append(points, [new\_x], axis=0)  
 values = np.append(values, [new\_condition], axis=0)  
  
 self.last\_points = np.array(points)  
 self.last\_values = np.array(values)  
 return np.array(points), np.array(values)  
  
def sweep\_method(matrix, d):  
 P = np.zeros((len(d),))  
 Q = np.zeros((len(d),))  
 n = len(d)  
 M = lambda i, j, inc: matrix[i+inc, j+inc]  
 a = lambda i: M(0, -1, i)  
 b = lambda i: M(0, 0, i)  
 c = lambda i: M(0, 1, i)  
 P[0] = -c(0) / b(0)  
 Q[0] = d[0] / b(0)  
 for i in range(1, n - 1):  
 P[i] = -c(i) / (b(i) + a(i) \* P[i - 1])  
 Q[i] = (d[i] - a(i) \* Q[i - 1]) / (b(i) + a(i) \* P[i - 1])  
 P[n - 1] = 0  
 Q[n - 1] = (d[n - 1] - a(n - 1) \* Q[n - 2]) / (b(n - 1) + a(n - 1) \* P[n - 2])  
 x = np.zeros(n)  
 x[n - 1] = Q[-1]  
 for i in range(n - 2, -1, -1):  
 x[i] = P[i] \* x[i + 1] + Q[i]  
 return x  
  
  
  
def increase\_matrix\_size(matrix, begin):  
 shape = matrix.shape  
 shape = tuple([i + 1 for i in shape])  
 temp = np.zeros(shape)  
 if begin:  
 temp[1:, 1:] = matrix  
 else:  
 temp[:-1,:-1] = matrix  
 return temp  
  
def increase\_vec\_size(vec, begin):  
 size = vec.shape[0] + 1  
 temp = np.zeros(size)  
 if begin:  
 temp[1:] = vec  
 else:  
 temp[:-1] = vec  
 return temp  
  
  
def solution(x):  
 return np.sin(x) + 2 - np.sin(x) \* np.log((1 + np.sin(x)) / (1 - np.sin(x)))  
  
def f(x):  
 return 0  
  
def p(x):  
 return np.tan(x)  
  
def q(x):  
 return 2  
  
  
  
*# уравнение xy'' - 2(x + 1)y + 2y = 0  
# так как x0 = 0, в точке x0 уравнение принимает вид  
# -y' + 2y = 0, что легко приводится к y = e^2x => y(0) = 1  
  
# cond1 = {"order":1, "value":1}  
# cond2 = {"order":2, "beta":-2, "value":-4}  
# cond2 = {"order":1, "value": 2 + np.exp(2)}*h = 0.01  
x0 = 0  
x1 = np.pi / 6  
y0 = 2  
y1 = 2.5 - 0.5 \* np.log(3)  
cond1 = {**"order"**: 1, **"value"**: y0}  
cond2 = {**"order"**: 1, **"value"**: y1}  
  
  
def finite\_differences(p, q, f, h, x0, x1, cond1, cond2):  
 points = np.arange(x0 + h, x1, h)  
 a = lambda x: (1 / np.power(h, 2) - p(x) / (2 \* h))  
 b = lambda x: -2 / np.power(h, 2) + q(x)  
 c = lambda x: (1 / np.power(h, 2) + p(x) / (2 \* h))  
 n = len(points)  
 matrix = np.zeros((n, n))  
 d = np.zeros((n,))  
 for i in range(0, n):  
 d[i] = f(points[i])  
  
 if i - 1 >= 0:  
 matrix[i,i - 1] = a(points[i])  
  
 matrix[i,i] = b(points[i])  
  
 if i + 1 < len(points):  
 matrix[i, i + 1] = c(points[i])  
  
 if cond1[**"order"**] == 1:  
 d[0] -= cond1[**"value"**] \* a(points[0])  
 else:  
 matrix = increase\_matrix\_size(matrix, True)  
 d = increase\_vec\_size(d, True)  
 matrix[1,0] = a(points[0])  
 matrix[0,0] = -(2 / (h \* (2 - p(x0) \* h))) + (q(x0) \* h) / (2 - p(x0) \* h) + cond1[**"alpha"**]  
 matrix[0,1] = 2 / (h \* (2 - p(x0) \* h))  
 d[0] = cond1[**"value"**] + (h \* f(x0)) / (2 - p(x0) \* h)  
 points = np.insert(points, 0, x0)  
  
 if cond2[**"order"**] == 1:  
 d[-1] -= cond2[**"value"**] \* c(points[-1])  
 else:  
 matrix = increase\_matrix\_size(matrix, False)  
 d = increase\_vec\_size(d, False)  
 matrix[-2, -1] = c(points[-1])  
 matrix[-1, -2] = - 2 / (h \* (2 + p(x1) \* h))  
 matrix[-1, -1] = (2 / (h \* (2 + p(x1) \* h))) - (q(x1) \* h) / (2 + p(x1) \* h) + cond2[**"beta"**]  
 d[-1] = cond2[**"value"**] - (h \* f(x1)) / (2 + p(x1) \* h)  
 points = np.insert(points, len(points), x1)  
  
 y = sweep\_method(matrix, d)  
 return points, y  
  
def f1(x,y1,y2):  
 return y2  
  
def f2(x,y1,y2):  
 return np.tan(y2) - 2 \* y1  
  
def secant(fun, x0, x1, epsilon=0.001):  
 fx0 = None  
  
 while True:  
 fx1 = fun(x1)  
 if fx0 is None:  
 fx0 = fun(x0)  
 new\_x0, new\_x1 = x1, x1 - (fx1 \* (x1 - x0))/(fx1 - fx0)  
 if np.abs(new\_x1 - new\_x0) < epsilon:  
 return new\_x1  
 x0, x1 = new\_x0, new\_x1  
 fx0 = fx1  
  
  
def shooting\_method(functions, x0, x1, y0, y1, tan1, h, epsilon):  
 def solve(x):  
 solver = RungeKutta(functions, x0, [y0, x])  
 points, values = solver.integrate(x1, h)  
 return values[-1, 0] - y1  
  
 result\_tan = secant(solve, tan1, tan1 + h / 2, epsilon)  
 p, y = RungeKutta(functions, x0, [y0, result\_tan]).integrate(x1, h)  
 return p, y[:,0]  
  
  
def fd\_error():  
 hp, hy = finite\_differences(p, q, f, h, x0, x1, cond1, cond2)  
 h2p, h2y = finite\_differences(p, q, f, h / 2, x0, x1, cond1, cond2)  
 error = [np.abs(hy[i] - h2y[2\*i]) for i in range(0, len(hp))]  
 error = np.array(error) / (np.power(2,2) - 1)  
 return error  
  
def sm\_error():  
 hp, hy = shooting\_method([f1, f2], x0, x1, y0, y1, 0.5, h, 0.001)  
 h2p, h2y = shooting\_method([f1, f2], x0, x1, y0, y1, 0.5, h / 2, 0.001)  
 error = [np.abs(hy[i] - h2y[2\*i]) for i in range(0, min([len(hp), len(h2p) // 2]))]  
 error = np.array(error) / (np.power(2, 2) - 1)  
 return error  
  
  
  
  
fd\_points, fd\_y = finite\_differences(p, q, f, h, x0, x1, cond1, cond2)  
precise = solution(fd\_points)  
sm\_points, sm\_y = shooting\_method([f1, f2], x0, x1, y0, y1, 0.5, h, 0.001)  
plt.plot(fd\_points, solution(fd\_points), label=**"precise solution"**)  
plt.plot(fd\_points, fd\_y, label=**"finite differences solution"**)  
plt.plot(sm\_points, sm\_y, label=**"shooting solution"**)  
print(**"finite differences error {}"**.format(fd\_error()))  
print()  
print(**"shooting method error {}"**.format(sm\_error()))  
print()  
print(**"finide differences precise error {}"**.format(np.abs(solution(fd\_points) - fd\_y)))  
print()  
print(**"shooting method precise error {}"**.format(np.abs(solution(sm\_points) - sm\_points)))  
  
  
  
plt.legend()  
plt.show()

Результат:

finite differences error [1.64361519e-03 1.61707510e-03 1.59005831e-03 1.56257972e-03

1.53465444e-03 1.50629778e-03 1.47752523e-03 1.44835246e-03

1.41879535e-03 1.38886992e-03 1.35859237e-03 1.32797907e-03

1.29704654e-03 1.26581145e-03 1.23429060e-03 1.20250096e-03

1.17045960e-03 1.13818372e-03 1.10569066e-03 1.07299785e-03

1.04012283e-03 1.00708324e-03 9.73896817e-04 9.40581389e-04

9.07154854e-04 8.73635188e-04 8.40040433e-04 8.06388692e-04

7.72698125e-04 7.38986935e-04 7.05273369e-04 6.71575708e-04

6.37912261e-04 6.04301358e-04 5.70761345e-04 5.37310577e-04

5.03967411e-04 4.70750197e-04 4.37677280e-04 4.04766982e-04

3.72037607e-04 3.39507425e-04 3.07194674e-04 2.75117548e-04

2.43294193e-04 2.11742701e-04 1.80481103e-04 1.49527366e-04

1.18899382e-04 8.86149669e-05 5.86918527e-05 2.91476813e-05]

shooting method error [0.00000000e+00 3.22530891e-05 6.53195486e-05 9.91577553e-05

1.33727541e-04 1.68990332e-04 2.04909238e-04 2.41449093e-04

2.78576470e-04 3.16259671e-04 3.54468692e-04 3.93175185e-04

4.32352411e-04 4.71975183e-04 5.12019817e-04 5.52464078e-04

5.93287133e-04 6.34469505e-04 6.75993040e-04 7.17840870e-04

7.59997396e-04 8.02448273e-04 8.45180405e-04 8.88181952e-04

9.31442353e-04 9.74952355e-04 1.01870407e-03 1.06269102e-03

1.10690826e-03 1.15135248e-03 1.19602211e-03 1.24091757e-03

1.28604141e-03 1.33139869e-03 1.37699722e-03 1.42284809e-03

1.46896618e-03 1.51537086e-03 1.56208693e-03 1.60914580e-03

1.65658706e-03 1.70446059e-03 1.75282952e-03 1.80177427e-03

1.85139854e-03 1.90183813e-03 1.95327471e-03 2.00595829e-03

2.06024609e-03 2.11667564e-03 2.17611760e-03 2.24015063e-03

2.31225544e-03]

finide differences precise error [4.00038570e-05 7.80758551e-05 1.12445324e-04 1.41503941e-04

1.63806850e-04 1.78073758e-04 1.83189990e-04 1.78207513e-04

1.62345930e-04 1.34993433e-04 9.57077310e-05 4.42169416e-05

1.95795516e-05 9.56102714e-05 1.83630852e-04 2.83223277e-04

3.93795142e-04 5.14578958e-04 6.44631471e-04 7.82833018e-04

9.27886909e-04 1.07831883e-03 1.23247628e-03 1.38852801e-03

1.54446353e-03 1.69809257e-03 1.84704460e-03 1.98876838e-03

2.12053149e-03 2.23941990e-03 2.34233749e-03 2.42600570e-03

2.48696303e-03 2.52156468e-03 2.52598208e-03 2.49620251e-03

2.42802861e-03 2.31707799e-03 2.15878273e-03 1.94838891e-03

1.68095611e-03 1.35135686e-03 9.54276095e-04 4.84210505e-04

6.45320707e-05 6.97833375e-04 1.42176548e-03 2.24259164e-03

3.16676712e-03 4.20094026e-03 5.35195346e-03 6.62684437e-03]

shooting method precise error [2. 1.99979983 1.99919867 1.9981955 1.99678933 1.99497917

1.992764 1.99014284 1.98711468 1.98367853 1.97983337 1.97557822

1.97091207 1.96583393 1.96034278 1.95443762 1.94811746 1.94138127

1.93422805 1.92665678 1.91866645 1.91025604 1.90142451 1.89217084

1.88249399 1.87239291 1.86186655 1.85091385 1.83953374 1.82772513

1.81548695 1.80281807 1.78971739 1.77618378 1.76221608 1.74781313

1.73297374 1.71769673 1.70198085 1.68582487 1.66922751 1.65218747

1.63470343 1.61677403 1.59839789 1.57957358 1.56029963 1.54057456

1.52039683 1.49976484 1.47867697 1.45713154 1.43512682 1.41266101]

