Московский Авиационный Институт

(национальный исследовательский университет)

**Факультет Прикладной математики и физики**

Кафедра Вычислительной математики и программирования

**Курсовая работа**

**по дисциплине**

**«Численные методы»**

**III курс, VI семестр**

Студент: Гришин А. Ю.

Группа: М8О-308Б-21

Руководитель: Ревизников Д. Л.

**Москва, 2024**

Содержание

[Лабораторная работа №1 5](#_Toc170684766)

[О программе 5](#_Toc170684767)

[Запуск программ 5](#_Toc170684768)

[Алгоритм LU-разложения матрицы 5](#_Toc170684769)

[Входные данные 7](#_Toc170684770)

[Реализация LU-разложения 8](#_Toc170684771)

[Реализация решения СЛАУ с помощью LU-разложения 9](#_Toc170684772)

[Результат 10](#_Toc170684773)

[Метод прогонки 11](#_Toc170684774)

[Входные данные 12](#_Toc170684775)

[Реализация решения СЛАУ с помощью метода прогонки 12](#_Toc170684776)

[Результат 13](#_Toc170684777)

[Итерационные методы решения СЛАУ 14](#_Toc170684778)

[Входные данные 15](#_Toc170684779)

[Реализация метода простых итераций 16](#_Toc170684780)

[Результат 18](#_Toc170684781)

[Метод вращений 19](#_Toc170684782)

[Входные данные 21](#_Toc170684783)

[Реализация метода вращений Якоби 21](#_Toc170684784)

[Результат 22](#_Toc170684785)

[QR-алгоритм нахождения собственных значений матриц 23](#_Toc170684786)

[Входные данные 24](#_Toc170684787)

[Реализация -разложения 25](#_Toc170684788)

[Результат 27](#_Toc170684789)

[Лабораторная работа №2 28](#_Toc170684790)

[О программе 28](#_Toc170684791)

[Запуск программ 28](#_Toc170684792)

[Решение нелинейных уравнений 28](#_Toc170684793)

[Входные данные 29](#_Toc170684794)

[Результат 31](#_Toc170684795)

[Решение систем нелинейных уравнений 31](#_Toc170684796)

[Входные данные 33](#_Toc170684797)

[Реализация метода простых итераций и Ньютона 34](#_Toc170684798)

[Результат 35](#_Toc170684799)

[Лабораторная работа №3 36](#_Toc170684800)

[Описание программы 36](#_Toc170684801)

[Запуск программ 36](#_Toc170684802)

[Интерполяция 36](#_Toc170684803)

[Входные данные 38](#_Toc170684804)

[Реализация интерполяционных многочленов Лагранжа и Ньютон 38](#_Toc170684805)

[Результат 39](#_Toc170684806)

[Сплайн-интерполяция 40](#_Toc170684807)

[Входные данные 41](#_Toc170684808)

[Реализация интерполяционных многочленов Лагранжа и Ньютона 41](#_Toc170684809)

[Результат 43](#_Toc170684810)

[Метод наименьших квадратов 43](#_Toc170684811)

[Входные данные 44](#_Toc170684812)

[Реализация метода наименьших квадратов 44](#_Toc170684813)

[Результат 45](#_Toc170684814)

[Численное дифференцирование 46](#_Toc170684815)

[Входные данные 47](#_Toc170684816)

[Реализация численного дифференцирования 47](#_Toc170684817)

[Результат 48](#_Toc170684818)

[Численное интегрирование 48](#_Toc170684819)

[Входные данные 50](#_Toc170684820)

[Реализация методов численного интегрирования 50](#_Toc170684821)

[Результат 51](#_Toc170684822)

[Лабораторная работа №4 52](#_Toc170684823)

[Описание программы 52](#_Toc170684824)

[Запуск программ 52](#_Toc170684825)

[Численные методы решения задачи Коши 52](#_Toc170684826)

[Входные данные 55](#_Toc170684827)

[Реализация методов численного решения задачи Коши 55](#_Toc170684828)

[Результат 57](#_Toc170684829)

[Численные методы решения краевой задачи для ОДУ 57](#_Toc170684830)

[Входные данные 60](#_Toc170684831)

[Реализация численных методов решения краевой задачи 60](#_Toc170684832)

[Результат 62](#_Toc170684833)

# Лабораторная работа №1

## О программе

Для выполнения данной лабораторной работы были написаны программы на языке программирования C++ для каждого из заданий. Для поддержки матричных операций был реализован класс TMatrix, объекты которого поддерживали операции сложения, матричного умножения матриц, чтение из файлов и форматированный вывод матрицы.

## Запуск программ

Для запуска программ необходимо скомпилировать исходный код путем ввода команды g++ lab-1/task\_{n}/main.cpp, где n – номер задания. После чего запустить скомпилированный файл.

## Алгоритм LU-разложения матрицы

‑разложение матрицы является методом решения задачи разложения матрицы в виде произведения нижней и верхней треугольных матриц , где — нижняя треугольная матрица, — верхняя треугольная матрица.

Одним из вариантов построения ‑разложения может быть использование метода Гаусса. Для понимания работы алгоритма рассмотрим -й шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов -го столбца матрицы . Элементы результирующей матрицы можно описать следующий образом:

Если выразить данный процесс в виде матричных операций, то увидим, что данная запись эквивалентна записи , где матрица определяется следующим образом:

Рассмотрим теперь, как можно выразить прямой ход метода Гаусса при помощи выше описанных матричных операций:

В результате прямого хода метода Гаусса получим , где — верхняя треугольная матрица. Соответствующая матрице матрица может быть выражена следующий образом . Если вычислить это выражение, то получим следующий вид матрицы

Полученное -разложение может быть эффективно использовано в задачах решения системы линейных алгебраических уравнений вида , в поиске детерминанта матрицы , поиске обратной матрицы .

Рассмотрим более подробно процесс решения линейных алгебраических уравнений с помощью -разложения.

1. Первым шагом является разложение матрицы коэффициентов на произведение матриц . В итоге, исходное уравнение можно представить в виде , или в виде системы
2. На втором шаге решается уравнение . Поскольку матрица системы – нижняя треугольная, решение можно записать в явном виде:
3. И, в заключении, решается система с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

Также, задачу поиска определителя матрицы можно легко свести к задаче -разложения. Разложив матрицу в виде произведения ее определитель можно найти по формуле:

Задачу поиска обратной матрицы также можно свести к задаче -разложения, если воспользоваться свойством . Это уравнение также можно решить методом ‑разложения.

### Входные данные

Ниже приведен пример входных данных для решения систему линейный алгебраических уравнений, соответствующий моему варианту



### Реализация LU-разложения

Ниже предоставлен исходный код, содержащий реализацию алгоритма -разложения. Также, стоит отметить, что данная реализация имеет оптимизацию в виде введения дополнительной матрицы перестановок , что применимо конкретно к выполнения алгоритма на ЭВМ.

std::pair<int, int> ForwardStep(Matrix::TMatrix& m, int k, std::vector<float>& coef) {

    int n = m.GetSize().first;

    // find element with maximum square to avoid small dividers

    int swapIndex = k;

    for (int i = k + 1; i < n; ++i) {

        float a = m.Get(swapIndex, k);

        float b = m.Get(i, k);

        if (a \* a > b \* b) {

            swapIndex = i;

        }

    }

    m.SwapRaws(k, swapIndex);

    // try to find a string with a non-zero element

    // and swap it with the current

    if (m.Get(k, k) == 0.0) {

        return {k, k};

    }

    // convert raws below so that the elements are zero

    for (int i = k + 1; i < n; ++i) {

        float c = m.Get(i, k) / m.Get(k, k);

        for (int j = k; j < n; ++j) {

            m.Set(i, j, m.Get(i, j) - c \* m.Get(k, j));

        }

        coef.push\_back(c);

    }

    return {k, swapIndex};

}

void LUDecompose(

const Matrix::TMatrix& a,

Matrix::TMatrix& l,

Matrix::TMatrix& u,

Matrix::TMatrix& p

) {

    int n = a.GetSize().first;

    std::vector<float> coef;

    p = Matrix::TMatrix::Eye(n);

    l = Matrix::TMatrix::Eye(n);

    u = a;

    coef.reserve(n); // reserve max possible number of bytes to avoid allocations

    for (int k = 0; k < n - 1; ++k) {

        std::pair<int, int> swap = ForwardStep(u, k, coef);

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

            l.Set(i + k + 1, k, coef[i]);

        }

        p.SwapRaws(swap.first, swap.second);

        coef.clear();

    }

}

### Реализация решения СЛАУ с помощью LU-разложения

Ниже представлен код, содеражщий реализацию алгоритма решения системы линейных алгебраических уравнений с помощью ранее описанного -разложения.

void SolveWithL(const Matrix::TMatrix& l, const Matrix::TMatrix& b, Matrix::TMatrix& x) {

    int n = l.GetSize().first;

    for (int i = 0; i < n; ++i) {

        float c = b.Get(i, 0);

        for (int j = 0; j < i; ++j) {

            c -= x.Get(j, 0) \* l.Get(i, j);

        }

        x.Set(i, 0, c);

    }

}

void SolveWithU(const Matrix::TMatrix& u, const Matrix::TMatrix& b, Matrix::TMatrix& x) {

    int n = u.GetSize().first;

    for (int i = n - 1; i >= 0; --i) {

        float c = b.Get(i, 0);

        for (int j = i + 1; j < n; ++j) {

            c -= x.Get(j, 0) \* u.Get(i, j);

        }

        c /= u.Get(i, i);

        x.Set(i, 0, c);

    }

}

void SolveSystem(

    const Matrix::TMatrix& l,

    const Matrix::TMatrix& u,

    const Matrix::TMatrix& p,

    const Matrix::TMatrix& b,

    Matrix::TMatrix& x

) {

    Matrix::TMatrix z(b.GetSize().first, 1);

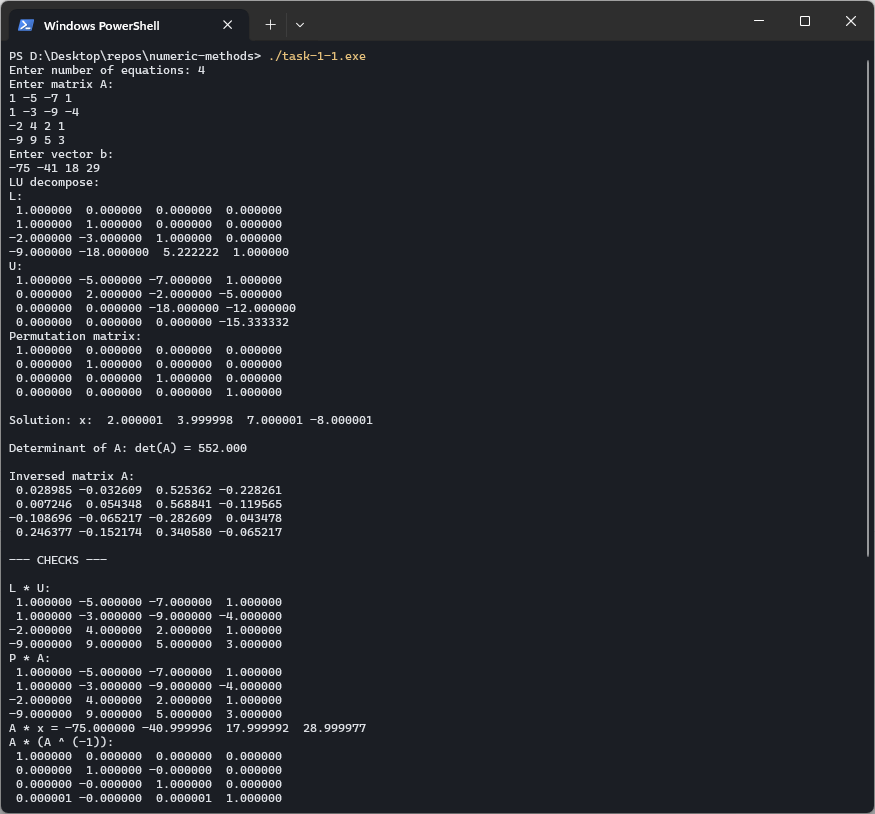
    SolveWithL(l, p \* b, z);

    SolveWithU(u, z, x);

}

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## Метод прогонки

Для случаев, когда в СЛАУ матрица коэффициентов имеет трехдиагональный вид:

может быть применен более оптимизированный вид метода Гаусса под названием метод прогонки. При использовании метода прогонки, решение системы происходит итеративно следующим образом

где – прогоночные коэффициенты, определяемые по формулам

После того как будут найдены все прогоночные коэффициенты, что соответствует этапу прямого хода метода прогонк, можно вычислить значения неизвестных путем обратной подстановки (обратный ход):

Для того, чтобы удостовериться, что данные подходят для решения методом прогонки, можно воспользоваться достаточным условием корректности и устойчивости к погрешностям вычислений является условие преобладания диагональных коэффициентов:

### Входные данные

Ниже приведен пример входных данных для решения систему линейный алгебраических уравнений, соответствующий моему варианту



### Реализация решения СЛАУ с помощью метода прогонки

Ниже предоставлен пример кода, содержащего реализацию метода прогонки и решения СЛАУ с трехдиагональной матрицей с помощью него

void CalculateRunCoefficients(const Matrix::TMatrix& A, const Matrix::TMatrix& B, Matrix::TMatrix& result) {

    int n = A.GetSize().first;

    for (int i = 0; i < n; ++i) {

        float a = A.Get(i, 0), b = A.Get(i, 1), c = A.Get(i, 2);

        float d = B.Get(i, 0);

        if (i == 0) {

            if (B.Get(i, 0) == 0.0) {

                throw std::runtime\_error("Can't find solution of system");

            }

            result.Set(i, 0, - c / b);

            result.Set(i, 1, d / b);

        }

        else {

            float PLast = result.Get(i - 1, 0), QLast = result.Get(i - 1, 1);

            float t = b + a \* PLast;

            if (t == 0.0) {

                throw std::runtime\_error("Can't find solution of system");

            }

            result.Set(i, 0, - c / t);

            result.Set(i, 1, (d - a \* QLast) / t);

        }

    }

}

void SolveUsingRunCoefficients(const Matrix::TMatrix& runCoefs, Matrix::TMatrix& result) {

    int n = runCoefs.GetSize().first;

    result.Set(n - 1, 0, runCoefs.Get(n - 1, 1));

    for (int i = n - 2; i >= 0; --i) {

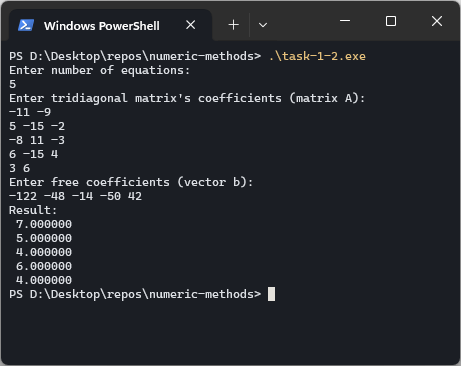
        result.Set(i, 0, runCoefs.Get(i, 0) \* result.Get(i + 1, 0) + runCoefs.Get(i, 1));

    }

}

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## Итерационные методы решения СЛАУ

Итерационные методы позволяют решать СЛАУ поэтапно. На каждом этапе решение рассматриваемой системы становится все более точным. Для дальнейшего рассмотрения итерационных методов, рассмотрим СЛАУ

где матрица коэффициентов является невырожденной.

Приведем СЛАУ к следующему виду путем эквивалентных преобразований

или в векторно-матричной форме . Мы получили общий вид решения системы уравнений с помощью итерационных методов

Однако, при таком определении мы сталкиваемся с проблемой бесконечного поиска решения СЛАУ. В этом случае применяют критерий окончания итерационного процесса. Например, им может послужить неравенство

В итоге, задача сводится к определению элементов . Одним из наиболее распространенных является следующий. Разрешим систему относительно неизвестных при ненулевых диагональных элементах (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением). Получим следующие выражения для компонентов вектора и матрицы эквивалентной системы:

В итоге мы пришли к определению одного из методов итераций – методу простых итераций. Его достаточным условием сходимости является диагональное преобладание матрицы по строкам или по столбцам:

Одним из недостатков метода простых итераций является его довольно медленная сходимость. Для решения этой проблемы используют **метод Зейделя**, заключающийся в том, что при вычислении компонента вектора неизвестных на (k+1)-й итерации используются , уже вычисленные на (k+1)-й итерации. Тогда метод Зейделя для известного вектора на k-ой итерации имеет вид:

### Входные данные

Ниже приведен пример входных данных для решения системы линейный алгебраических уравнений с использованием методов итераций, соответствующий моему варианту



### Реализация метода простых итераций

Ниже предоставлена реализация метода простых реализаций на языке программирования C++

struct IterativeMethodResult {

    Matrix::TMatrix result;

    int iterations;

};

float Norm(const Matrix::TMatrix& m) {

    float matrixSum = 0.0;

    float element;

    int n = m.GetSize().first;

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            element = m.Get(i, j);

            matrixSum += element \* element;

        }

    }

    return std::sqrt(matrixSum);

}

void JakobiMethod(const Matrix::TMatrix& A, const Matrix::TMatrix& b, Matrix::TMatrix& alpha, Matrix::TMatrix& beta) {

    int n = A.GetSize().first;

    float diagAlpha;

    for (int i = 0; i < n; ++i) {

        diagAlpha = A.Get(i, i);

        beta.Set(i, 0, b.Get(i, 0) / diagAlpha);

        for (int j = 0; j < n; ++j) {

            alpha.Set(

                i, j,

                (i == j) ? 0.0 : - A.Get(i, j) / diagAlpha

            );

        }

    }

}

void inverseMatrix(const Matrix::TMatrix& m, Matrix::TMatrix& result) {

    int n = m.GetSize().first;

    Matrix::TMatrix l(n, n), u(n, n), p(n, n);

    LUDecompose(m, l, u, p);

    InverseMatrix(l, u, p, result);

}

void SeidelMethod(const Matrix::TMatrix& A, const Matrix::TMatrix& b, Matrix::TMatrix& alpha, Matrix::TMatrix& beta) {

    int n = A.GetSize().first;

    Matrix::TMatrix B(n, n), C(n, n), T(n, n);

    Matrix::TMatrix E = Matrix::TMatrix::Eye(n);

    JakobiMethod(A, b, alpha, beta);

    // split alpha matrix B, C: alpha = B + C

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            if (j >= i) C.Set(i, j, alpha.Get(i, j));

            else B.Set(i, j, alpha.Get(i, j));

        }

    }

    inverseMatrix(E + B \* (-1.0), T);

    alpha = T \* C;

    beta = T \* beta;

}

float Epsilon(float alphaNorm, const Matrix::TMatrix& x1, const Matrix::TMatrix& x2) {

    float coef = alphaNorm / (1 - alphaNorm);

    Matrix::TMatrix diff = x1 + x2 \* (-1.0);

    return coef \* Norm(diff);

}

IterativeMethodResult IterativeMethod(const Matrix::TMatrix& alpha, const Matrix::TMatrix& beta, float eps) {

    int n = alpha.GetSize().first;

    float alphaNorm = Norm(alpha);

    int iterations = 0;

    Matrix::TMatrix x = beta;

    Matrix::TMatrix x2 = beta;

    while (true) {

        iterations++;

        x2 = alpha \* x + beta;

        if (Epsilon(alphaNorm, x, x2) <= eps) {

            break;

        }

        x = x2;

    }

    return {

        .result=x2,

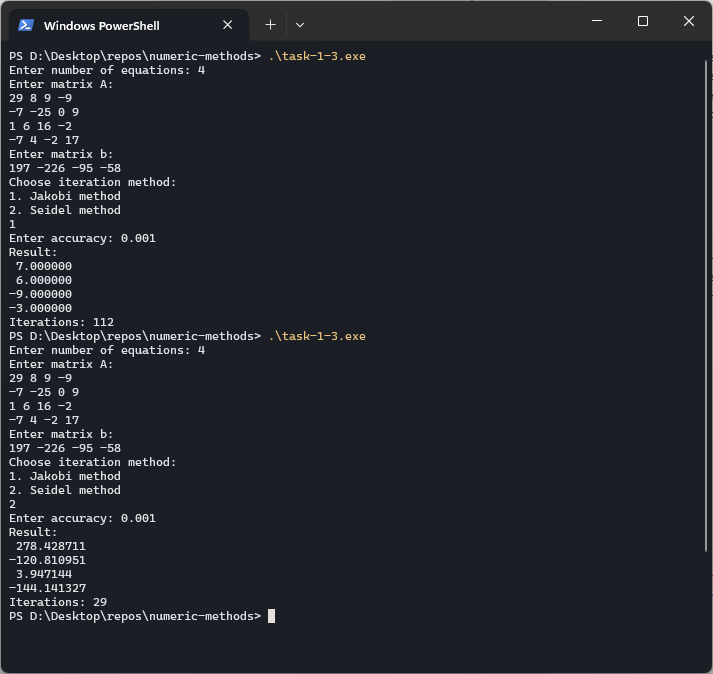
        .iterations=iterations

    };

}

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## Метод вращений

Среди численных методов для решения задачи поиска собственных значений матрицы существует метода вращений Якоби. Данный метод имеет ограничения на входящую матрицу в виде симметричности (). Его задачей является решение полной проблемы собственных значений и собственных векторов матриц.

Данный метод основывается на идее итерационном применении преобразований подобия

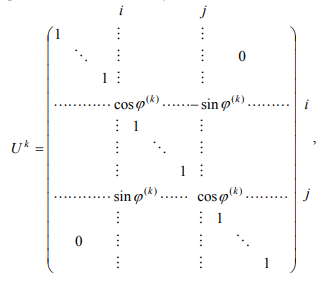
поскольку для симметрических матриц матрица преобразования подобия является ортогональной (), то , где – диагональная матрица с собственными значениями на главной диагонали.

Сформулируем алгоритм работы методы вращений Якоби. Пусть дана симметрическая матрица . Требуется вычислить все собственные значения и соответствующие им собственные векторы с заданной точностью .

Пусть известна матрица на k–й итерации, при этом для : .

1. На начальной итерации (при ) имеем ;
2. На каждой итерации выбирается максимальный по модулю недиагональный элемент матрицы ;

1. Ставится задача найти такую ортогональную матрицу , чтобы в результате преобразования подобия произошло обнуление элемента матрицы . В качестве ортогональной матрицы выбирается матрица вращения, имеющая следующий вид:



Угол вращения определяется из условия :

причем если , то .

1. Строится матрица

Так как метод вращений Якоби также является итерационным, то возникает необходимость в определении критерия окончания итерационного процесса. Он имеет следующий вид:

Данное неравенство описывает условие малости суммы квадратов в недиагональных элементах.

### Входные данные

Ниже приведен пример входных данных для решения задачи поиска собственных значений с использованием метода вращений Якоби, соответствующий моему варианту



### Реализация метода вращений Якоби

Ниже предоставлена программная реализация метода вращения Якоби

std::pair<int, int> FindMaxNotDiagonalElement(const Matrix::TMatrix& A) {

    int n = A.GetSize().first;

    int maxI = 0, maxJ = 1;

    for (int i = 0; i < n; ++i) {

        for (int j = i + 1; j < n; ++j) {

            if (fabsf(A.Get(i, j)) > fabsf(A.Get(maxI, maxJ))) {

                maxI = i;

                maxJ = j;

            }

        }

    }

    return {maxI, maxJ};

}

float GetRotationAngle(const Matrix::TMatrix& A, int i, int j) {

    float aii = A.Get(i, i), ajj = A.Get(j, j), aij = A.Get(i, j);

    if (aii == ajj) return M\_PI / 4.0;

    return 0.5 \* atan((2.0 \* aij) / (aii - ajj));

}

Matrix::TMatrix GetRotationMatrix(int n, int i, int j, float phi) {

    Matrix::TMatrix U = Matrix::TMatrix::Eye(n);

    float c = cos(phi), s = sin(phi);

    U.Set(i, i, c);

    U.Set(j, j, c);

    U.Set(i, j, -s);

    U.Set(j, i, s);

    return U;

}

Matrix::TMatrix GetNextA(const Matrix::TMatrix& A, const Matrix::TMatrix& U) {

    return Matrix::Transpose(U) \* A \* U;

}

float t(const Matrix::TMatrix& A) {

    int n = A.GetSize().first;

    float s = 0.0, el;

    for (int i = 0; i < n; ++i) {

        for (int j = i + 1; j < n; ++j) {

            el = A.Get(i, j);

            s += el \* el;

        }

    }

    return sqrt(s);

}

EigenTaskResult SolveEigenTask(const Matrix::TMatrix& M, float eps) {

    Matrix::TMatrix A(M);

    int n = A.GetSize().first;

    Matrix::TMatrix U = Matrix::TMatrix::Eye(n);

    EigenTaskResult result;

    result.iterations = 0;

    while (t(A) > eps) {

        std::pair<int, int> pos = FindMaxNotDiagonalElement(A);

        float phi = GetRotationAngle(A, pos.first, pos.second);

        Matrix::TMatrix u = GetRotationMatrix(n, pos.first, pos.second, phi);

        A = GetNextA(A, u);

        U = U \* u;

        result.iterations++;

    }

    for (int i = 0; i < n; ++i) {

        result.eigenValues.push\_back(A.Get(i, i));

    }

    for (int i = 0; i < n; ++i) {

        Matrix::TMatrix v(n, 1);

        for (int j = 0; j < n; ++j) {

            v.Set(j, 0, U.Get(j, i));

        }

        result.eigenVectors.push\_back(v);

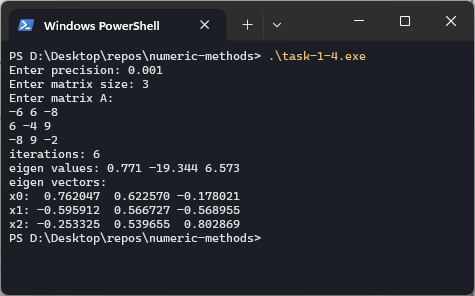
    }

    return result;

}

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## QR-алгоритм нахождения собственных значений матриц

Также, для решения задачи поиска собственных значений и собственных векторов матрицы может быть применен метод QR-разложения. Его суть заключается в представлении исходной матрицы в виде следующего произведения

где – ортогональная матрица, а – верхняя треугольная. Одним из возможных способов построения разложения является использование преобразования Хаусхолдера, который позволяет обратить в нуль группу поддиагональных элементов столбца матрицы.

Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

где – произвольный ненулевой столбец.

Далее, для решения поставленной задачи, необходимо рассмотреть случай, когда необходимо обратить в нуль все элементы вектора кроме первого. Для этого необходимо построить матрицу Хаусхолдера такую, чтобы она удовлетворяла следующему равенству

где – столбец , в котором все элементы, кроме первого, стали нулевыми.

В этом случае вектор имеет следующий вид:

Применяя описанную процедуру для обнуления поддиагональных элементов каждого столбца исходной матрицы, можно за фиксированное количество шагов выполнить ее QR-разложение.

Процедура QR-разложения многократно применяется в QR-алгоритме для вычисления собственных значений. При этом строится следующий итерационный процесс:

Таким образом, каждая итерация выполняется в два этапа. На первом этапе матрица разлагается на произведение ортогональной матрицы и верхней треугольной матрицы . На втором этапе эти матрицы перемножаются в обратном порядке.

Если у матрицы нет кратных собственных значений, последовательность сходится к верхней треугольной матрице (если все собственные значения вещественные) или к верхней квазитреугольной матрице (если имеются комплексно-сопряженные пары собственных значений).

Таким образом, каждому вещественному собственному значению будет соответствовать столбец с поддиагональными элементами, стремящимися к нулю. В качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать следующее неравенство:

### Входные данные

Ниже приведен пример входных данных для решения задачи поиска собственных значений с использованием метода -разложения, соответствующий моему варианту



### Реализация -разложения

Ниже предоставлена программная реализация алгоритма -разложения. Она также включает в себя вычисление матрицы Хаусхолдера.

using ComplexPair = std::pair<std::complex<float>, std::complex<float>>;

using EigenValues = std::vector<std::complex<float>>;

using ChangeHistory = std::vector<float>;

const int HISTORY\_SIZE = 5;

void GetHouseholderMatrix(const Matrix::TMatrix& A, int i, Matrix::TMatrix& H) {

    int n = A.GetSize().first;

    Matrix::TMatrix v(n, 1);

    for (int j = 0; j < n; ++j) {

        if (j < i) {

            v.Set(j, 0, 0);

        }

        else if (j > i) {

            v.Set(j, 0, A.Get(j, i));

        }

        else {

            float aDiag = A.Get(i, i);

            float signA = float((aDiag > 0) - (aDiag < 0));

            float sum = 0.0;

            for (int t = j; t < n; ++t) {

                sum += A.Get(t, i) \* A.Get(t, i);

            }

            v.Set(

                j, 0,

                aDiag + signA \* sqrt(sum)

            );

        }

    }

    float k = (Matrix::Transpose(v) \* v).Get(0, 0);

    H = v \* Matrix::Transpose(v) \* (-2.0 / k) + Matrix::TMatrix::Eye(n);

}

void QRDecompose(const Matrix::TMatrix& A, Matrix::TMatrix& Q, Matrix::TMatrix& R) {

    int n = A.GetSize().first;

    Matrix::TMatrix H(n, n);

    Q = Matrix::TMatrix::Eye(n);

    R = A;

    for (int i = 0; i < n - 1; ++i) {

        GetHouseholderMatrix(R, i, H);

        Q = Q \* H;

        R = H \* R;

    }

}

ComplexPair FindComplexEigeValues(const Matrix::TMatrix& A, int i) {

    float a1 = A.Get(i, i), a2 = A.Get(i + 1, i + 1);

    float a3 = A.Get(i + 1, i), a4 = A.Get(i, i + 1);

    float b = - a1 - a2;

    float c = a1 \* a2 - a3 \* a4;

    float d = b \* b - 4.0 \* c;

    std::complex<float> dSqrt = std::sqrt(std::complex<float>(d, 0));

    std::complex<float> bComplex = std::complex<float>(b, 0.0);

    std::complex<float> k = 0.5;

    return { k \* (-bComplex + dSqrt), k \* (-bComplex - dSqrt) };

}

bool tReal(const Matrix::TMatrix& A, int i, int j, float eps) {

    int n = A.GetSize().first;

    float sum = 0.0;

    for (int t = j; t < n; ++t) {

        sum += A.Get(t, i) \* A.Get(t, i);

    }

    return std::sqrt(sum) <= eps;

}

float tComplex(const Matrix::TMatrix& Ai, int i, float eps) {

    int n = Ai.GetSize().first;

    Matrix::TMatrix Q(n, n), R(n, n);

    QRDecompose(Ai, Q, R);

    Matrix::TMatrix ANext = R \* Q;

    ComplexPair lambda1 = FindComplexEigeValues(Ai, i);

    ComplexPair lambda2 = FindComplexEigeValues(ANext, i);

    return (std::abs(lambda2.first - lambda1.first) <= eps) && (std::abs(lambda2.second - lambda1.second) <= eps);

}

bool IsEigenValueReal(const ChangeHistory& history) {

    int startIndex = std::max(1, int(history.size()) - HISTORY\_SIZE);

    for (int i = startIndex; i < history.size(); ++i) {

        if (history[i] >= history[i - 1]) {

            return false;

        }

    }

    return true;

}

void UpdateChangeHistory(const Matrix::TMatrix& A, std::vector<ChangeHistory>& history) {

    int n = A.GetSize().first;

    for (int i = 0; i < n - 1; ++i) {

        history[i].push\_back(std::abs(A.Get(i + 1, i)));

    }

}

EigenValues GetEigenValues(const Matrix::TMatrix& A, float eps) {

    int n = A.GetSize().first;

    Matrix::TMatrix Q(n, n), R(n, n);

    Matrix::TMatrix Ai = A;

    EigenValues values;

    std::vector<std::vector<float>> history(n);

    int i = 0;

    while (i < n) {

        QRDecompose(Ai, Q, R);

        Ai = R \* Q;

        UpdateChangeHistory(Ai, history);

        if (tReal(Ai, i, i + 1, eps)) {

            values.push\_back(Ai.Get(i, i));

            i++;

        } else if (tComplex(Ai, i, eps) && tReal(Ai, i, i + 2, eps)) {

            ComplexPair p = FindComplexEigeValues(Ai, i);

            values.push\_back(p.first);

            values.push\_back(p.second);

            i += 2;

        }

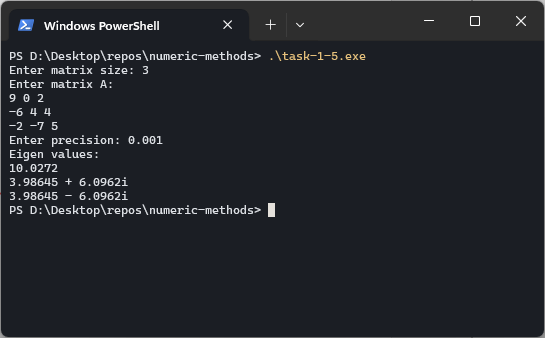
    }

    return values;

}

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту

**

# Лабораторная работа №2

## О программе

Для решения данной лабораторной работы были написаны программы на языке Python с использованием библиотек numpy и PyQt для графического интерфейса.

## Запуск программ

Для запуска программ необходимо установить перечисленные выше библиотеки и выполнить команду python –m lab\_2.task\_{n}.app, где n – номер задания в лабораторной работе.

## Решение нелинейных уравнений

Рассмотрим задачу решения трансцендентного уравнения . Используя метод простой итерации, можно эффективно находить корни этого уравнения. Предположим, что корень находится на отрезке , и функция удовлетворяет следующим условиям:

1. на концах отрезка функция имеет разные знаки;
2. первая и вторая производные имеют постоянные знаки.

Для построения итерационного процесса уравнение заменяется эквивалентным уравнением , где – непрерывная функция. Выберем начальное приближение и организуем итерационный процесс по следующей схеме:

Условие для всех гарантирует сходимость итерационного процесса, если начальное приближение выбрано вблизи корня.

Также, для решения этой же задачи можно воспользоваться методом Ньютона. Для построения итерационного процесса этим методом функция , непрерывная на интервале , должна удовлетворять условиям, аналогичным тем, что применяются в методе итераций.

Правило построения итерационной последовательности основано на замене нелинейной функции её линейной моделью, используя формулу Тейлора. Выберем две соседние точки в окрестности корня, так что , и запишем разложение функции в ряд Тейлора:

Полагая, что и оставляя только линейную часть разложения, получаем соотношение метода Ньютона:

Используя условие сходимости метода простых итераций, можно получить достаточное условие сходимости метода Ньютона в форме:

Для выбора начального приближения используется теорема, согласно которой в качестве начального приближения нужно выбрать тот конец интервала, где знак функции совпадает со знаком второй производной:

Для завершения итерационного процесса используется следующий критерий:

### Входные данные

Ниже приведен пример входных данных для решения задачи поиска решения уравнения вида с использованием методов Ньютона и метода простых итераций, соответствующий моему варианту

Реализация метода простых итераций и Ньютона

Ниже предоставлена программная реализация метода простых итераций и Ньютона

class MethodResult(NamedTuple):

    x: float

    iterations: int

def \_derivative(f: Function) -> Function:

    DX = 0.00001

    return lambda x: (f(x + DX) - f(x)) / DX

def \_sign(x: float) -> float:

    if x == 0.0:

        return 0.0

    if x < 0.0:

        return -1.0

    return 1.0

def \_build\_phi(f: Function, a: float, b: float) -> Function:

    df = \_derivative(f)

    max\_df = max\_value(lambda x: abs(df(x)), a, b)

    sign\_f = \_sign(df(a))

    return lambda x: x - (sign\_f / max\_df) \* f(x)

def iterations\_method(f: Function, a: float, b: float, eps: float, iterations: int) -> MethodResult:

    x0 = (a + b) / 2.0

    phi = \_build\_phi(f, a, b)

    dphi = \_derivative(phi)

    q = max\_value(lambda x: abs(dphi(x)), a, b)

    last\_x = x0

    i = 0

    while i <= iterations:

        x = phi(last\_x)

        if abs(x - last\_x) <= (1 - q) / q \* eps:

            return MethodResult(x, i)

        last\_x = x

        i += 1

    return MethodResult(last\_x, i)

def newton\_method(f: Function, a: float, b: float, eps: float, iterations: int) -> MethodResult:

    x0 = (a + b) / 2.0

    last\_x = x0

    df = \_derivative(f)

    i = 0

    while i <= iterations:

        x = last\_x - f(last\_x) / df(last\_x)

        if abs(x - last\_x) <= eps:

            return MethodResult(x, i)

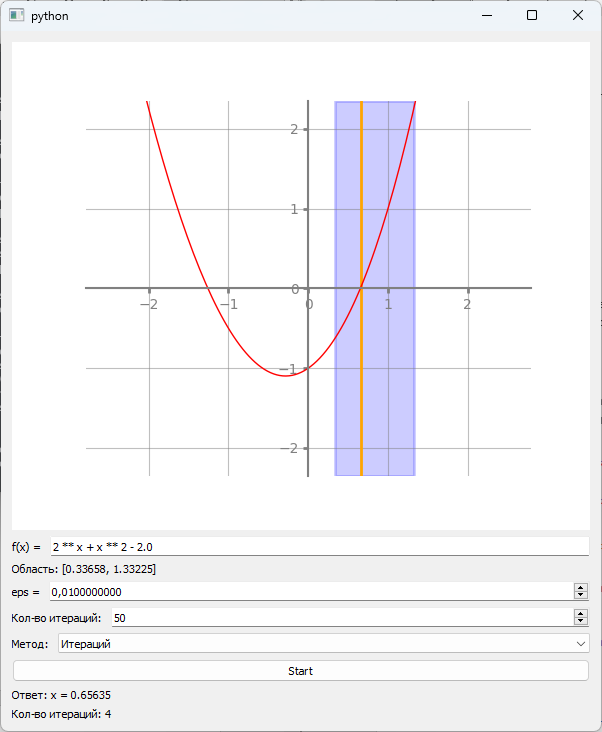
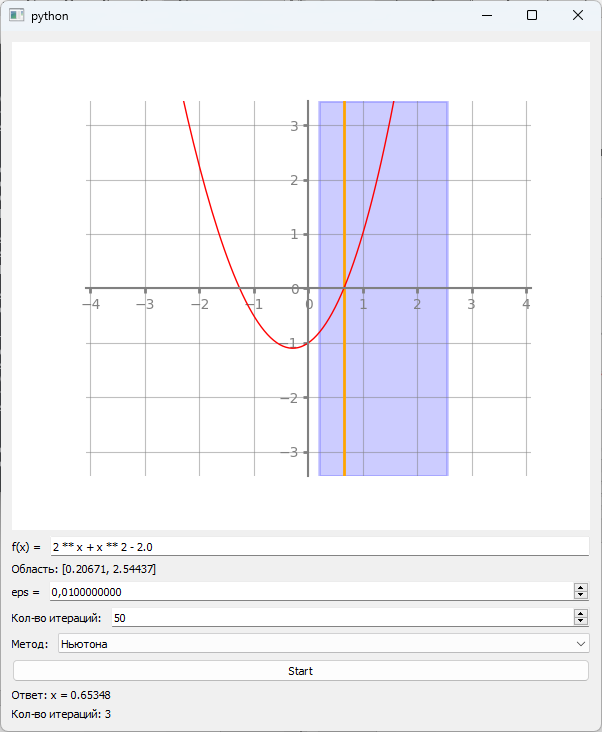
        last\_x = x

        i += 1

    return MethodResult(last\_x, i)

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту

## Решение систем нелинейных уравнений

Задача решения системы нелинейных уравнений можно описать в виде решения следующей системы

где все функции системы непрерывны и дифференцируемы вблизи решения.

Аналогично с решением одного уравнения, используя метод простых итераций, мы можем записать уравнение в следующий эквивалентный вид

Тогда, на основе начального приближения можно построить решение системы нелинейных уравнений методом простых итераций по следующей схеме

На практике используют следующий критерий окончания итерации для данного метода

Это условие описывает то, что метод простых итераций сходится к решению, если какая-либо норма матрицы Якоби , построенная по правым частям эквивалентной системы в замкнутой области , меньше единицы на каждой итерации.

Также, часто используют матричную форму этого критерия

Или, если вычисление этого критерия затруднительно:

Как и в случае с решением одного уравнения, метод Ньютона отличается от метода простых итераций определением перехода к следующему решению. В метод Ньютона он имеет вид:

где – матрица Якоби, имеющая вид:

В практических вычислениях в качестве условия окончания итераций обычно используют критерий, выполняющийся для всех переменных системы:

### Входные данные

Ниже приведен пример входных данных для решения задачи поиска решения системы нелинейных уравнений с использованием методов Ньютона и метода простых итераций, соответствующий моему варианту



где значение параметра .

### Реализация метода простых итераций и Ньютона

Ниже предоставлена программная реализация метода простых итераций и Ньютона для решения системы нелинейных уравнений

class MethodResult(NamedTuple):

    x: np.ndarray[float]

    iterations: int

class VectorFunction(np.ndarray):

    def \_\_new\_\_(cls, input\_array) -> "VectorFunction":

        obj = np.asarray(input\_array).view(cls)

        return obj

    def \_\_call\_\_(self, x: Vector) -> Vector:

        return np.array([func(x) for func in self])

def \_sign(x: float) -> float:

    if x == 0.0:

        return 0.0

    if x < 0.0:

        return -1.0

    return 1.0

def \_partial\_derivative(f: MultiArgFunction, arg\_index: int) -> MultiArgFunction:

    dx = 0.0001

    def \_df(x):

        n = len(x)

        x2 = x + np.array([dx if i == arg\_index else 0.0 for i in range(n)])

        return (f(x2) - f(x)) / dx

    return \_df

def \_derivative(f: MultiArgFunction, n: int) -> VectorFunction:

    return VectorFunction([\_partial\_derivative(f, i) for i in range(n)])

def \_jakobi\_matrix(f: VectorFunction) -> VectorFunction:

    n = len(f)

    return VectorFunction([\_derivative(el, n) for el in f])

def \_norm(matrix: Vector) -> float:

    return abs(matrix).max()

def \_build\_phi(f: VectorFunction, n: int, index: int, s1: Vector, s2: Vector) -> MultiArgFunction:

    f\_el = f[index]

    df = \_derivative(f\_el, n)

    pdf = \_partial\_derivative(f\_el, index)

    f\_sign = \_sign(pdf(s1))

    mx = max\_value(lambda x: \_norm(df(x)), s1, s2)

    return lambda x: x[index] - (f\_sign / mx) \* f\_el(x)

def \_solve\_system(A: Matrix, b: Vector) -> Vector:

    l, u, p = task\_1.lu\_decompose(A)

    return task\_1.solve\_system(l, u, p, b)

def iteration\_method(f: MultiArgFunction, s1: Vector, s2: Vector, eps: float, iterations: int) -> MethodResult:

    n = len(f)

    phi = VectorFunction([\_build\_phi(f, n, i, s1, s2) for i in range(n)])

    last\_x = (s1 + s2) / 2.0

    i = 0

    while i <= iterations:

        x = phi(last\_x)

        if \_norm(x - last\_x) <= eps:

            return MethodResult(x, i)

        last\_x = x

        i += 1

    return MethodResult(last\_x, i)

def newton\_method(f: MultiArgFunction, s1: Vector, s2: Vector, eps: float, iterations: int) -> MethodResult:

    J = \_jakobi\_matrix(f)

    last\_x = (s1 + s2) / 2.0

    i = 0

    while i <= iterations:

        dx = \_solve\_system(J(last\_x), -f(last\_x))

        x = last\_x + dx

        if \_norm(x - last\_x) <= eps:

            return MethodResult(x, i)

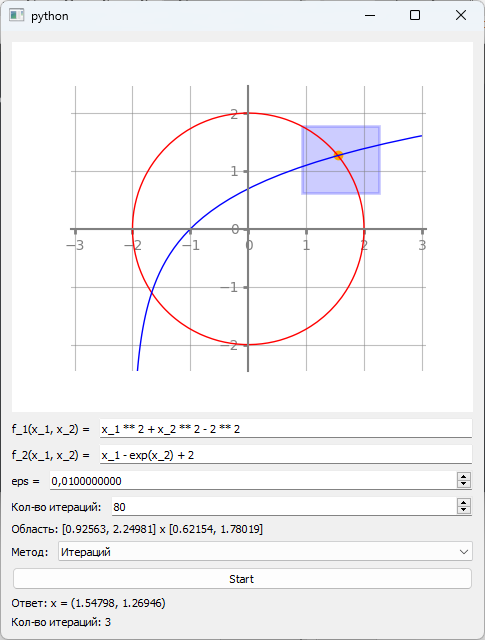
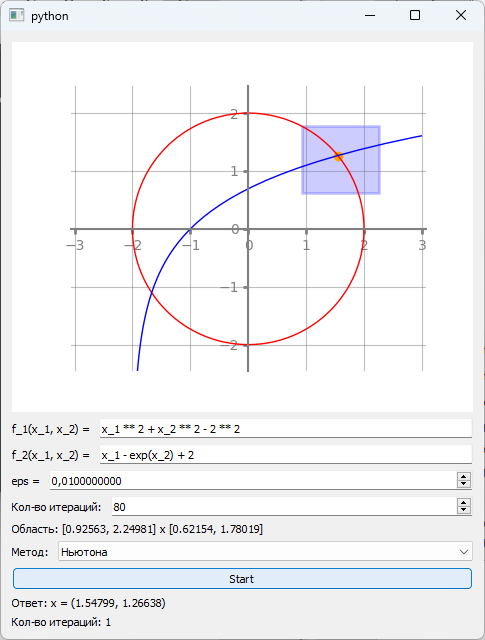
        last\_x = x

        i += 1

    return MethodResult(last\_x, i)

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту

# Лабораторная работа №3

## Описание программы

Для решения текущей лабораторной работы была написана программа на языке Python с применением таких библиотек как scipy, numpy, PyQt и matplotlib.

## Запуск программ

Для запуска программ необходимо установить все вышеперечисленные библиотеки и ввести команду python –m lab\_3.task\_{n}, где n – номер задания лабораторной работы.

## Интерполяция

Рассмотрим множество различных точек на отрезке , где известны значения функции . Приближающая функция , для которой выполняются условия , называется

интерполяционной функцией.

Чаще всего в качестве такой функции используют многочлены степени , имеющие вид:

Произвольный многочлен можно представить следующим образом:

где —многочлены степени , которые называются лагранжевыми многочленами влияния. Они удовлетворяют условию:

Лагранжевые многочлены влияния определяются как:

Таким образом, интерполяционный многочлен Лагранжа имеет вид:

Основным недостатком многочлена Лагранжа является необходимость пересчета всех коэффициентов при добавлении новых интерполяционных узлов. Чтобы избежать этого недостатка, используют интерполяционный многочлен Ньютона.

Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются и определяются через разделенные разности нулевого порядка:

Разделенные разности второго порядка определяются через разности первого порядка:

Разделенная разность -го порядка определяется рекурсивно:

Интерполяционный многочлен Ньютона имеет вид:

При добавлении новых узлов, начальные члены многочлена Ньютона остаются неизменными. Для повышения точности интерполяции могут быть добавлены новые члены, что требует подключения дополнительных интерполяционных узлов, причём порядок их добавления не имеет значения. Это является преимуществом многочлена Ньютона по сравнению с многочленом Лагранжа.

### Входные данные

Ниже приведен пример входных данных для решения задачи интерполяции функции с использованием интерполяционных многочленов Лагранжа и Ньютона, соответствующий моему варианту.

; 

a) ; б) 

Реализация интерполяционных многочленов Лагранжа и Ньютона

Ниже предоставлена программная реализация интерполяционных многочленов Лагранжа и Ньютона. Данные методы интерполяции были представлены в виде классов, которые принимают на вход координаты узлов и поддерживают интерфейс вызываемых (Callable) объектов для того, чтобы их можно было использовать как функции.

class LagrangeInterpolationPolynomial:

    def \_\_init\_\_(self, nodes: list[tuple[float, float]]):

        self.\_x, self.\_y = zip(\*nodes) if nodes else ([], [])

    def l(self, i: int, x: float) -> float:

        value = 1.0

        for j in range(len(self.\_x)):

            if j == i:

                continue

            value \*= (x - self.\_x[j]) / (self.\_x[i] - self.\_x[j])

        return value

    def \_\_call\_\_(self, x: float) -> float:

        value = 0.0

        for i in range(len(self.\_x)):

            value += self.\_y[i] \* self.l(i, x)

        return value

class NewtonInterpolationPolynomial:

    def \_\_init\_\_(self, nodes: list[tuple[float, float]]):

        x, y = zip(\*nodes) if nodes else ([], [])

        self.\_diffs = self.\_count\_differences(x, y)

        self.\_x = x

    @staticmethod

    def \_count\_differences(x: list[float], y: list[float]) -> list[list[float]]:

        n = len(x)

        diffs = [[] for \_ in range(n)]

        for i in range(n):

            if i == 0:

                diffs[0] = y

            else:

                for j in range(n - i):

                    diffs[i].append(

                        (diffs[i - 1][j] - diffs[i - 1][j + 1]) / (x[j] - x[j + i])

                    )

        return diffs

    def \_\_call\_\_(self, x: float) -> float:

        value = 0.0

        for i in range(len(self.\_x)):

            coef = 1.0

            for j in range(i):

                coef \*= x - self.\_x[j]

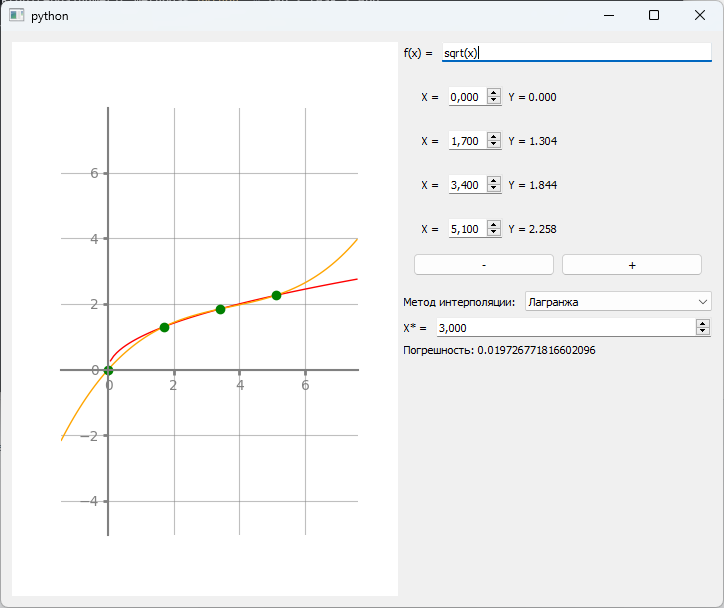
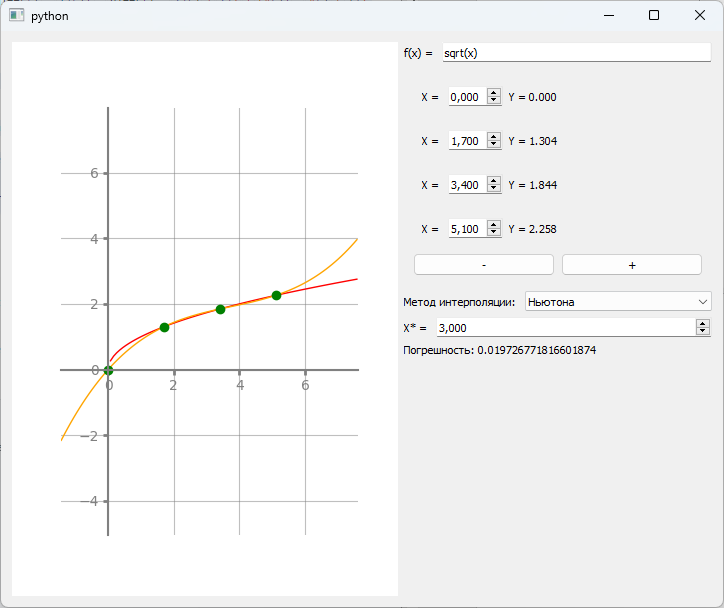
            coef \*= self.\_diffs[i][0]

            value += coef

        return value

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту

## Сплайн-интерполяция

Применение одной интерполяционной формулы на большом количестве узлов может оказаться неэффективным. Интерполяционный многочлен может проявлять значительные колебания, и его значения между узлами могут сильно отклоняться от значений самой функции. Для решения этой проблемы используется сплайн-интерполяция.

Суть метода заключается в использовании различных интерполяционных функций для непересекающихся промежутков, с обеспечением стыковки значений функции и её производных на границах этих промежутков.

Наиболее распространённым является случай, когда между двумя соседними точками отрезка строится многочлен степени :

В узлах интерполяции этот многочлен принимает значения аппроксимируемой функции и остаётся непрерывным вместе со своими производными до порядка . Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Коэффициенты сплайна определяются из условий равенства в узлах значений сплайна и исходной функции, а также равенства производных до порядка .

На практике часто используется кубический сплайн, который удобно записать в виде:

где .

Для построения кубического сплайна необходимо определить многочленов третьей степени, что требует нахождения неизвестных коэффициентов . Эти коэффициенты вычисляются на основе условий в узловых точках.

### Входные данные

Ниже приведен пример входных данных для решения задачи интерполяции функции с использованием Сплайн-интерполяции, соответствующий моему варианту.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 0.0 | 1.7 | 3.4 | 5.1 | 6.8 |
|  | 0.0 | 1.3038 | 1.8439 | 2.2583 | 2.6077 |

3.0

### Реализация интерполяционных многочленов Лагранжа и Ньютона

Ниже предоставлена программная реализация Сплайн-интерполяции. Данный метод интерполяции был представлен в виде класса, который принимает на вход координаты узлов и поддерживает интерфейс вызываемых (Callable) объектов для того, чтобы его можно было использовать как функцию.

class CubicSplineInterploation:

    def \_\_init\_\_(self, nodes: list[tuple[float, float]]):

        x, f = map(np.array, zip(\*nodes))

        h = self.\_get\_h(x)

        self.c = self.\_get\_coef\_c(h, f)

        self.a = self.\_get\_coef\_a(f)

        self.b = self.\_get\_coef\_b(h, self.c, f)

        self.d = self.\_get\_coef\_d(h, self.c)

        self.x = x

    def \_\_call\_\_(self, x: float) -> float:

        for i in range(1, len(self.x)):

            if self.x[i - 1] <= x <= self.x[i]:

                h = x - self.x[i - 1]

                return self.a[i] + self.b[i] \* h + self.c[i] \* h\*\*2 + self.d[i] \* h\*\*3

        return np.nan

    @classmethod

    def \_get\_h(cls, x):

        n = len(x) - 1

        return np.array(

            [0] + [x[i] - x[i - 1] for i in range(1, n + 1)],

            dtype=np.float32,

        )

    @classmethod

    def \_get\_coef\_c(cls, h, f):

        A, B = cls.\_get\_system\_coefs(h, f)

        c = np.zeros\_like(h)

        c[2:] = solve\_system(A, B)

        return c

    @classmethod

    def \_get\_coef\_a(cls, f):

        a = np.zeros\_like(f)

        a[1:] = f[:-1]

        return a

    @classmethod

    def \_get\_coef\_b(cls, h, c, f):

        n = len(f) - 1

        b = np.zeros(n + 1)

        for i in range(1, n):

            b[i] = (f[i] - f[i - 1]) / h[i] - (h[i] \* (c[i + 1] + 2 \* c[i])) / 3.0

        b[n] = (f[n] - f[n - 1]) / h[n] - 2.0 / 3.0 \* h[n] \* c[n]

        return b

    @classmethod

    def \_get\_coef\_d(cls, h, c):

        n = len(c) - 1

        d = np.zeros(n + 1)

        for i in range(1, n):

            d[i] = (c[i + 1] - c[i]) / (3.0 \* h[i])

        d[n] = -c[n] / (3.0 \* h[n])

        return d

    @classmethod

    def \_get\_system\_coefs(cls, h, f):

        n = len(f) - 1

        A = np.zeros((n - 1, 3))

        B = np.zeros((n - 1, 1))

        for i in range(2, n + 1):

            a, b, c = h[i - 1], 2 \* (h[i - 1] + h[i]), h[i]

            d = 3 \* ((f[i] - f[i - 1]) / h[i] - (f[i - 1] - f[i - 2]) / h[i - 1])

            if i == 2:

                A[i - 2, :] = 0, b, c

            elif i == n:

                A[i - 2, :] = a, b, 0

            else:

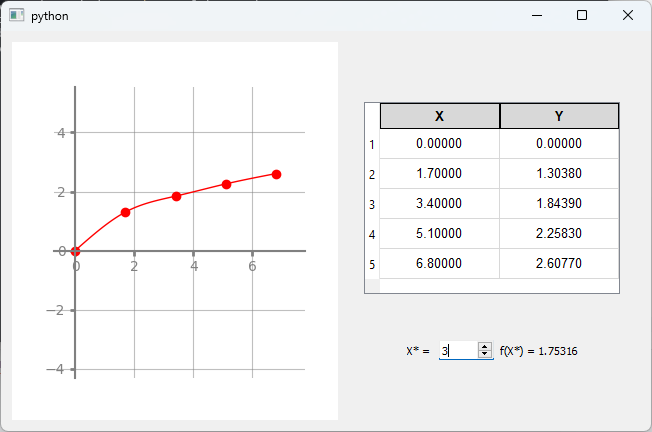
                A[i - 2, :] = a, b, c

            B[i - 2, 0] = d

        return A, B

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## Метод наименьших квадратов

Метод наименьших квадратов является одним из численных методов решения задачи аппроксимации функций. Предположим, у нас есть набор точек , где значения имеют некоторую погрешность. Наша задача - найти многочлен , который наилучшим образом приближает эти точки.

Мы знаем, что может быть представлена как многочлен степени с неизвестными коэффициентами . Для нахождения этих коэффициентов мы стремимся минимизировать квадратичное отклонение между значениями многочлена и фактическими значениями . Это приводит к функции ошибки , которую мы хотим минимизировать:

Чтобы найти минимум функции , мы используем необходимое условие экстремума, которое приводит нас к нормальной системе уравнений метода наименьших квадратов:

Эта система представляет собой линейную систему уравнений относительно коэффициентов , которую мы можем решить, чтобы построить многочлен , наилучшим образом приближающий функцию и минимизирующий квадратичное отклонение.

### Входные данные

Ниже приведен пример входных данных для решения задачи аппроксимации функции с использованием метода наименьших квадратов, соответствующий моему варианту.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | 0.0 | 0.2 | 0.4 | 0.6 | 0.8 | 1.0 |
|  | 1.0 | 1.0032 | 1.0512 | 1.2592 | 1.8192 | 3.0 |

### Реализация метода наименьших квадратов

Ниже предоставлена программная реализация метода наименьших квадратов. Метод наименьших квадратов здесь представлен в виде класса, конструктор которого принимает на вход последовательность точек-узлов. Данный класс реализует интерфейс вызываемых (Callable) объектов поэтому может быть в дальнейшем использован как функция.

class MinimalSquareInterpolation:

    def \_\_init\_\_(self, n: int, nodes: list[tuple[float, float]]) -> None:

        A, b = self.\_get\_normal\_system\_coefs(nodes, n)

        l, u, p = task\_1.lu\_decompose(A)

        self.\_coef = task\_1.solve\_system(l, u, p, b)

    @classmethod

    def \_get\_normal\_system\_coefs(cls, nodes, n):

        A = np.zeros((n + 1, n + 1))

        b = np.zeros(n + 1)

        for k in range(n + 1):

            for i in range(n + 1):

                A[k, i] = cls.\_get\_a\_coef(nodes, k, i)

            b[k] = cls.\_get\_b\_coef(nodes, k)

        return A, b

    @classmethod

    def \_get\_a\_coef(cls, nodes, k, i):

        value = 0.0

        N = len(nodes)

        for j in range(N):

            value += nodes[j][0] \*\* (k + i)

        return value

    @classmethod

    def \_get\_b\_coef(cls, nodes, k):

        value = 0.0

        N = len(nodes)

        for j in range(N):

            value += nodes[j][1] \* nodes[j][0] \*\* k

        return value

    def \_\_call\_\_(self, x: float) -> float:

        value = 0.0

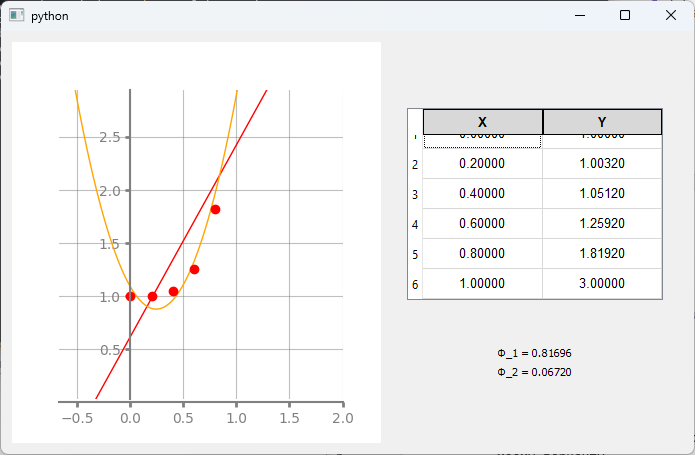
        for i, coef in enumerate(self.\_coef):

            value += coef \* (x\*\*i)

        return value

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## Численное дифференцирование

Задача численного дифференцирования заключается в нахождении приближенного значения производной функции , заданной таблично.

В численном дифференцировании мы заменяем исходную функцию на интерполяционный многочлен , где - набор коэффициентов, различный для каждого отрезка, и - остаточный член приближения. Это позволяет нам приближенно находить производные функции, полагая

На практике часто используются аппроксимации первой и второй производных. Первая производная может быть аппроксимирована отрезками прямой:

При использовании интерполяционного многочлена второй степени для аппроксимации имеем:

При равноотстоящих точках разбиения данная формула обеспечивает второй порядок точности. Для вычисления второй производной также необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем:

Таким образом, численное дифференцирование позволяет нам приближенно находить производные функции, основываясь на интерполяции исходных данных.

### Входные данные

Ниже приведен пример входных данных для решения задачи численного дифференцирования функции, соответствующий моему варианту.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -0.2 | 0.0 | 0.2 | 0.4 | 0.6 |
|  | 1.7722 | 1.5708 | 1.3694 | 1.1593 | 0.9273 |

 0.2

### Реализация численного дифференцирования

Ниже предоставлена программная реализация численного дифференцирования для нахождения производных первого и второго порядка с использованием интерполяционных многочленов различных порядков.

def find\_interval(X: Vector, x: float) -> Iterator[int]:

    return [i for i in range(len(X) - 1) if X[i] <= x <= X[i + 1]]

def first\_derivative(X: Vector, Y: Vector, x: float) -> float | tuple[float, float]:

    indexes = find\_interval(X, x)

    if not indexes:

        return np.nan

    results = []

    for i in indexes:

        numerator = Y[i + 1] - Y[i]

        denominator = X[i + 1] - X[i]

        results.append(numerator / denominator)

    if len(results) == 1:

        return results[0]

    return (results[0], results[1])

def second\_derivative(X: Vector, Y: Vector, x: float) -> float:

    indexes = find\_interval(X, x)

    if not indexes:

        return np.nan

    i = indexes[0]

    if i + 2 >= len(X):

        return np.nan

    term1 = (Y[i + 1] - Y[i]) / (X[i + 1] - X[i])

    term2 = (Y[i + 2] - Y[i + 1]) / (X[i + 2] - X[i + 1]) - term1

    term3 = 2 \* x - X[i] - X[i + 1]

    return term1 + term2 / (X[i + 2] - X[i]) \* term3

def second\_derivative\_factor(X: Vector, Y: Vector, x: float) -> float:

    indexes = find\_interval(X, x)

    if not indexes:

        return np.nan

    i = indexes[0]

    if i + 2 >= len(X):

        return np.nan

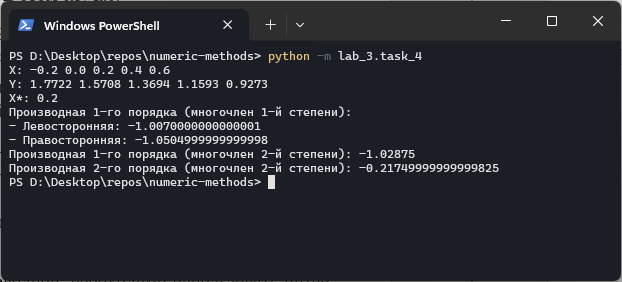
    term1 = (Y[i + 2] - Y[i + 1]) / (X[i + 2] - X[i + 1])

    term2 = (Y[i + 1] - Y[i]) / (X[i + 1] - X[i])

    return 2.0 \* (term1 - term2) / (X[i + 2] - X[i])

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



## Численное интегрирование

Численные методы интегрирования находят применение, когда аналитически вычислить определенный интеграл

не представляется возможным. Один из наиболее распространенных методов - замена подынтегральной функции на интерполяционный многочлен.

Используя интерполяционные многочлены разной степени, можно получить формулы интегрирования разной точности. Например, заменяя функцию на многочлен Лагранжа нулевой степени, проходящий через середину отрезка, мы получаем формулу прямоугольников:

Для таблично заданных функций удобно использовать интерполяционный многочлен первой степени, выбирая начало и конец отрезка. Это приводит к формуле трапеций:

Для увеличения точности можно заменить кривую параболой, используя интерполяционный многочлен второй степени. Получим формулу Симпсона:

Метод Рунге-Ромберга-Ричардсона позволяет улучшить точность вычислений. При наличии результатов вычислений на сетках с разными шагами и можно использовать следующую формулу:

Этот метод позволяет получить более точные результаты, основываясь на интерполяции и вычислении разницы между результатами на разных сетках.

### Входные данные

Ниже приведен пример входных данных для решения задачи численного интегрирования функции на отрезке , соответствующий моему варианту.





### Реализация методов численного интегрирования

Ниже предоставлена программная реализация методов численного интегрирования, такие как мтеод трапеций, прямоугольников, метод Симпсона. Методы представлены в виде функций, которые принимают на вход интегрируемую функцию и множество точек.

def rectangle\_method(f: Function, X: Vector) -> float:

    n = len(X)

    h = np.array([X[i] - X[i - 1] for i in range(1, n)])

    value = 0.0

    for i in range(n - 1):

        x = (X[i] + X[i + 1]) \* 0.5

        value += h[i] \* f(x)

    return value

def trapezoid\_method(Y: Vector, X: Vector) -> float:

    n = len(X)

    h = np.array([X[i] - X[i - 1] for i in range(1, n)])

    value = 0.0

    for i in range(n - 1):

        value += (Y[i] + Y[i + 1]) \* h[i]

    return value \* 0.5

def simpson\_method(f: Function, X: Vector) -> float:

    n = len(X)

    h = np.array([X[i] - X[i - 1] for i in range(1, n)])

    value = 0.0

    Y = np.zeros(2 \* n - 1)

    for i in range(n):

        Y[2 \* i] = f(X[i])

        if i + 1 < n:

            Y[2 \* i + 1] = f((X[i] + X[i + 1]) \* 0.5)

    for i in range(n - 1):

        value += (Y[2 \* i] + 4 \* Y[2 \* i + 1] + Y[2 \* i + 2]) \* h[i] \* 0.5

    return value / 3.0

def runge\_rombert\_method(integral\_1: float, h1: float, integral\_2: float, h2: float, p: int) -> tuple[float, float]:

    return (

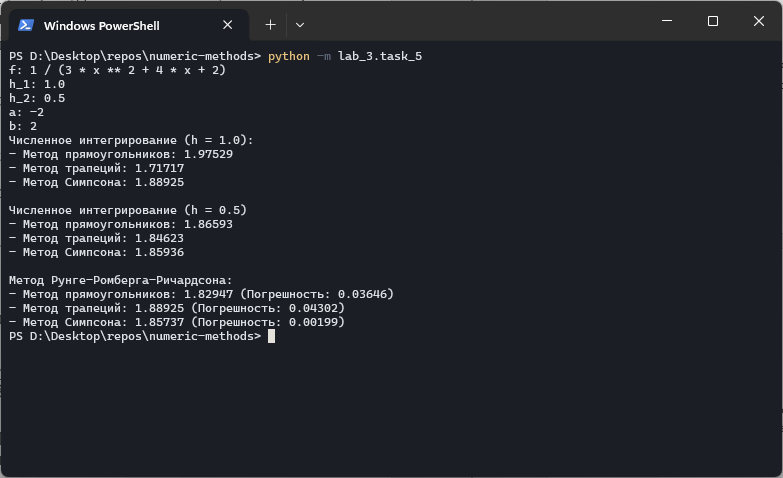
        integral\_1 + (integral\_1 - integral\_2) / ((h2 / h1) \*\* p - 1.0,

        abs((integral\_1 - integral\_2) / (2\*\*p - 1.0)),

    )

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту



# Лабораторная работа №4

## Описание программы

Для решения текущей лабораторной работы была написана программа на языке Python с применением таких библиотек как scipy, numpy, PyQt и matplotlib.

## Запуск программ

Для запуска программ необходимо установить все вышеперечисленные библиотеки и ввести команду python –m lab\_4.task\_{n}, где n – номер задания лабораторной работы.

## Численные методы решения задачи Коши

Задача Коши представляет из себя следующую систему

где – дифференциальное уравнение, а – начальное условие задачи Коши.

Для решения данной задачи с использованием численных методов часто вводят разностную сетку на отрезке :

Здесь, точки ​ - узлы сетки, - шаг разностной сетки.

Чтобы численно найти приближенное решение задачи Коши, мы будем использовать сеточную функцию, которая представляет совокупность значений функции на узлах сетки.

Явный метод Эйлера – один из методов численного решения обыкновенных дифференциальных уравнений. Его применение в практике ограничено из-за невысокой точности. Этот метод можно вывести разными способами: геометрически, используя разложение в ряд Тейлора, разностную аппроксимацию производной или квадратурный метод.

Для геометрического вывода метода Эйлера начнем с известного решения в точке из начального условия. Рассмотрим процесс нахождения решения в точке . График функции , являющейся решением задачи Коши, представляет собой гладкую кривую, проходящую через точку с касательной в этой точке. Угловой коэффициент касательной равен значению производной в , которое равно значению правой части дифференциального уравнения в этой точке.

При небольшом шаге график функции и касательная почти не различаются, так что мы можем использовать значение касательной вместо точного значения .

Погрешность представляет собой отрезок графически. Из прямоугольного треугольника мы получаем . Подставляя и производную из уравнения, мы получаем

Повторяя этот процесс для как начальной точки, мы получаем значение в точке и т.д.

Неявный метод Эйлера, или метод трапеций, - это численный метод решения задачи Коши для обыкновенного дифференциального уравнения первого порядка. Этот метод является классическим и отличается высокой точностью и устойчивостью при больших значениях шага интегрирования.

Основная идея метода заключается в замене производной функции на приближенное значение, используя формулу для среднего значения. В результате получается неявное уравнение, которое численно решается. Это позволяет учесть влияние будущих значений функции при вычислении текущего шага, что обеспечивает более точные результаты, особенно при интегрировании на больших интервалах.

Семейство явных методов Рунге-Кутты -го порядка записывается в общем виде через следующие формулы

где . В частности, метод Рунге-Кутты четвертого порядка имеет вид:

метод Адамса четвертого порядка точности основывается на использовании интерполяционного многочлена третьей степени, построенного по значениям подынтегральной функции в последних четырех узлах. Полученная формула для выражается следующим образом:

Метод Адамса, как и другие многошаговые методы, требует предварительного вычисления решений в первых четырех узлах. Решение в узле известно из начальных условий, а решения в узлах могут быть получены с использованием одношагового метода, например, метода Рунге-Кутты четвертого порядка.

Для системы обыкновенных дифференциальных уравнений первого порядка задача Коши имеет вид:

с начальными условиями . Для этой системы уравнений можно применить все вышеупомянутые методы, решая уравнения по порядку.

Для задачи Коши второго порядка можно преобразовать ее в систему уравнений, используя замену :

с начальными условиями , .

### Входные данные

Ниже приведен пример входных данных для решения задачи численного решения задачи Коши, соответствующий моему варианту.

где и размер шага . Аналитическое решение такой задачи Коши представляет из себя функцию вида

### Реализация методов численного решения задачи Коши

Ниже представлена программная реализация численных методов решения задачи Коши. Методы реализованы в виде функций, принимающих на вход данные системы и спроектированы таким образом, чтобы могли быть переиспользованы как для решения уравнений первого порядка, так и для второго.

def euler(eq: DiffEquation, y\_0: Vector, grid: Grid) -> Matrix:

    F = convert\_to\_system\_function(eq)

    X = grid.range

    Y = y\_0

    answer = np.zeros((1 + len(y\_0), len(X)))

    answer[:, 0] = (X[0], \*y\_0.flat)

    for i in range(1, len(X)):

        Y = Y + grid.h \* F(X[i - 1], Y)

        answer[:, i] = (X[i], \*Y.flat)

    return answer

def euler\_with\_continuations(eq: DiffEquation, y\_0: Vector, grid: Grid) -> Matrix:

    P = euler(eq, y\_0, grid)

    F = convert\_to\_system\_function(eq)

    X = grid.range

    Y = y\_0

    answer = np.zeros((1 + len(y\_0), len(X)))

    answer[:, 0] = (X[0], \*y\_0.flat)

    for i in range(1, len(X)):

        Y\_P = np.array(list(P[1:, i].flat))

        Y = Y + grid.h \* (F(X[i - 1], Y) + F(X[i], Y\_P)) \* 0.5

        answer[:, i] = (X[i], \*Y.flat)

    return answer

def runge\_kutta(eq: DiffEquation, y\_0: Vector, grid: Grid) -> Matrix:

    h = grid.h

    F = convert\_to\_system\_function(eq)

    X = grid.range

    Y = y\_0

    answer = np.zeros((1 + len(y\_0), len(X)))

    answer[:, 0] = (X[0], \*y\_0.flat)

    for i in range(1, len(X)):

        K\_1 = h \* F(X[i - 1], Y)

        K\_2 = h \* F(X[i - 1] + h \* 0.5, Y + K\_1 \* 0.5)

        K\_3 = h \* F(X[i - 1] + h \* 0.5, Y + K\_2 \* 0.5)

        K\_4 = h \* F(X[i - 1] + h, Y + K\_3)

        Y = Y + (K\_1 + 2.0 \* K\_2 + 2.0 \* K\_3 + K\_4) / 6.0

        answer[:, i] = (X[i], \*Y.flat)

    return answer

def adams(eq: DiffEquation, y\_0: Vector, grid: Grid, underlying\_method: OdeSolveMethod = runge\_kutta) -> Matrix:

    F = convert\_to\_system\_function(eq)

    X = grid.range

    h = grid.h

    answer = np.zeros((1 + len(y\_0), len(X)))

    answer[:, :4] = underlying\_method(eq, y\_0, Grid(X[0], X[3], h))

    for i in range(4, len(X)):

        f = [F(answer[0, j], answer[1:, j]) for j in range(i - 4, i)]

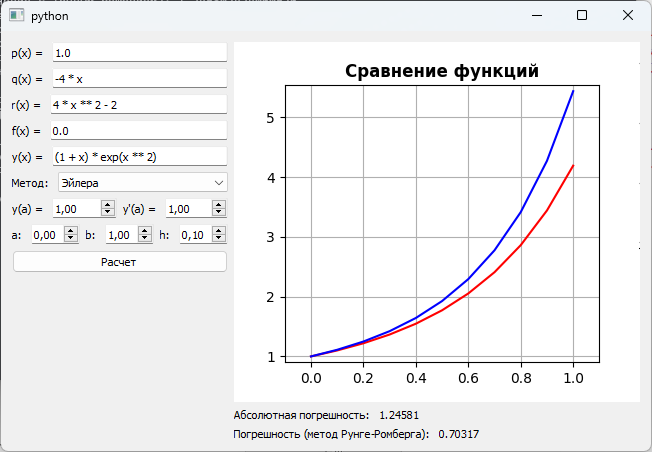
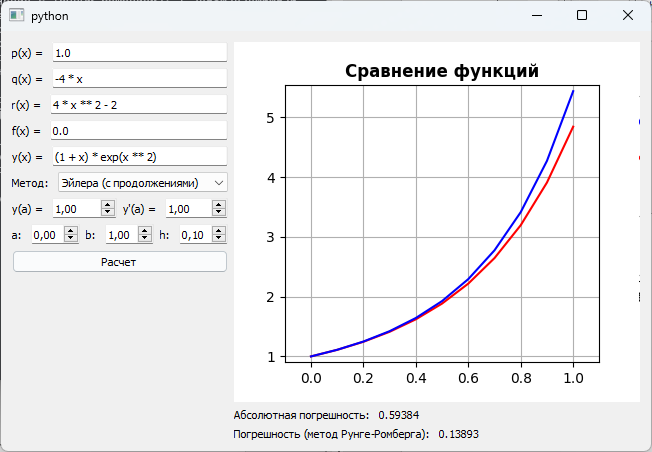
        Y = answer[1:, i - 1] + h / 24 \* (55 \* f[3] - 59 \* f[2] + 37 \* f[1] - 9 \* f[0])

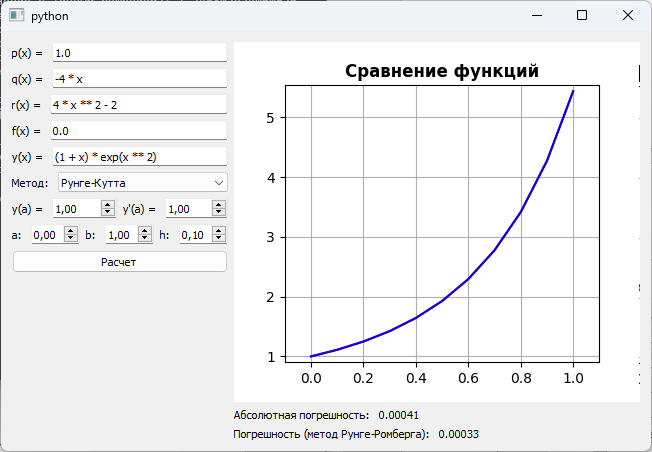
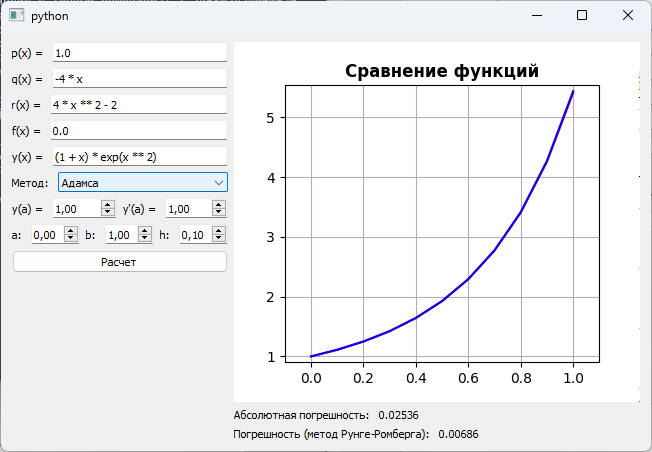
        answer[:, i] = (X[i], \*Y.flat)

    return answer

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту

## Численные методы решения краевой задачи для ОДУ

Краевая задача включает в себя два типа граничных условий. Граничные условия первого рода задают значения функции на концах отрезка, например

Граничные условия второго рода определяют значения производных функции на концах отрезка, например

или линейную комбинацию значений и производных, такую как

Возможно использование различных типов условий на разных концах отрезка.

Метод стрельбы – один из численных методов решения краевой задачи, который сводит задачу краевых условий к последовательному решению задачи Коши. Для начала, когда требуется решить краевую задачу с граничными условиями первого рода на отрезке , мы формулируем задачу Коши с уравнением

и начальными условиями и

Далее мы устанавливаем начальное значение параметра и решаем задачу Коши с помощью какого-либо метода. Если является решением этой задачи на интервале , мы сравниваем значение функции с требуемым значением в правом конце отрезка.

Затем мы сталкиваемся с задачей коррекции угла наклона касательной к решению в левом конце отрезка. Наша задача сформулирована как нахождение такого значения переменной , при котором решение в правом конце отрезка совпадает со значением . Иными словами, мы стремимся найти корень уравнения .

Следующее значение искомого корня определяется рекурсивно по формуле:

Такой подход позволяет приблизиться к искомому значению параметра с каждой итерацией, пока не будет достигнута необходимая точность.

Для дальнейшего описания метода конечных разностей рассмотрим задачу о двухточечной краевой задаче для линейного дифференциального уравнения второго порядка на отрезке :

Для численного решения данной краевой задачи применим метод конечных разностей. Вводятся следующие аппроксимации для производных:

Здесь представляет собой шаг дискретизации. Подставляя эти аппроксимации в дифференциальное уравнение, получаем систему уравнений для нахождения значений :

для , где .

### Входные данные

Ниже приведен пример входных данных для решения задачи численного решения краевой задачи, соответствующий моему варианту.

Аналитическое решение такой задачи имеет вид

### Реализация численных методов решения краевой задачи

Ниже приведена программная реализация метода стрельбы и метода конечных разностей. Методы реализованы в виде функций и спроектированы таким образом, чтобы вспомогательные методы, которые используют данные алгоритмы, передавались в качестве аргументов

def shooting\_method(

    eq: DiffEquation,

    cond\_1: BoundaryCondition,

    cond\_2: BoundaryCondition,

    grid: Grid,

    underlying\_method: OdeSolveMethod = runge\_kutta,

    found\_root=newton\_method,

) -> Matrix:

    def build\_ode\_condition(nu: float) -> Vector:

        match cond\_1:

            case (0.0, b, c):

                return np.array([-c / b, nu])

            case (a, 0.0, c):

                return np.array([nu, -c / a])

            case (a, b, c):

                return np.array(

                    [

                        -c / b - a / b \* nu,

                        nu,

                    ]

                )

    def count\_difference(y: float, dy: float):

        a, b, c = cond\_2

        return a \* dy + b \* y + c

    def g(nu: float) -> float:

        Y\_0 = build\_ode\_condition(nu)

        answer = underlying\_method(eq, Y\_0, grid)

        return count\_difference(answer[1][-1], answer[2][-1])

    nu = found\_root(g, 0.0, 4.0, 0.001, 10).x

    Y\_0 = build\_ode\_condition(nu)

    return underlying\_method(eq, Y\_0, grid)

def finite\_diff\_method(eq: DiffEquation, cond\_1: BoundaryCondition, cond\_2: BoundaryCondition, grid: Grid) -> Matrix:

    X = grid.range

    N = len(X)

    answer = np.zeros\_like(X)

    p = lambda x: eq.q(x) / eq.p(x)

    q = lambda x: eq.r(x) / eq.p(x)

    f = lambda x: - eq.f(x) / eq.p(x)

    h = grid.h

    g\_1 = lambda x: 1 - 0.5 \* p(x) \* h

    g\_2 = lambda x: q(x) \* h \*\* 2 - 2

    g\_3 = lambda x: 1 + 0.5 \* p(x) \* h

    g\_4 = lambda x: f(x) \* h \*\* 2

    c1, c2, c3 = cond\_1

    if c1 == 0.0:

        y\_0 = - c3 / c2

        first = [0.0, g\_2(X[1]), g\_3(X[1])], g\_4(X[1]) - g\_1(X[1]) \* y\_0

        i = 2

        answer[0] = y\_0

    elif c2 == 0.0:

        first = [0.0, -c1, c1], - c3 \* h

        i = 1

    else:

        first = [0.0, -c1, c1 + c2 \* h], - c3 \* h

        i = 1

    c1, c2, c3 = cond\_2

    if c1 == 0.0:

        y\_n = - c3 / c2

        last = [g\_1(X[-2]), g\_2(X[-2]), 0.0], g\_4(X[-2]) - g\_3(X[-2]) \* y\_n

        j = N - 2

        answer[N - 1] = y\_n

    elif c2 == 0.0:

        last = [-c1, c1, 0.0], - c3 \* h

        j = N - 1

    else:

        last = [-c1, c1 + c2 \* h, 0.0], - c3 \* h

        j = N - 1

    A = np.zeros((j - i + 2, 3)) # матрица коэффициентов (n x 3)

    B = np.zeros((j - i + 2, 1)) # матрица совбодных коэффициентов (n x 1)

    A[0, :], B[0, 0] = first

    A[-1, :], B[-1, 0] = last

    for t in range(i, j):

        t2 = t - i + 1

        x = X[t - 1]

        A[t2, :] = [g\_1(x), g\_2(x), g\_3(x)]

        B[t2, 0] = g\_4(x)

    # решаем систему методом прогонки

    answer[i - 1:j + 1] = solve\_system(A, B)

    return np.array([X, answer.flat])

### Результат

Ниже предоставлен пример работы программы с входными данными, соответствующими моему варианту

