**Numeral Methods**

***Completed: Mishanya03***

***2023***

Оглавление

[**Поиск корня уравнения скалярной функции** 3](#_Toc135411979)

[**Метод половинного деления** 3](#_Toc135411980)

[**Метод Ньютона** 3](#_Toc135411981)

[**Метод последовательного приближения** 3](#_Toc135411982)

[**Матричные методы** 3](#_Toc135411983)

[**Методы – верхний и нижний треугольники** 3](#_Toc135411984)

[**Метод Гаусса** 4](#_Toc135411985)

[**Метод квадратных корней** 4](#_Toc135411986)

[**Метод Грамма-Шмидта** 4](#_Toc135411987)

[**Метод последовательных приближений** 4](#_Toc135411988)

[**Сплайн функции(интерполяция)** 5](#_Toc135411989)

[**Метод наименьших квадратов (МНК)** 5](#_Toc135411990)

[**Вычисление определенного интеграла разными способами** 6](#_Toc135411991)

[**Метод прямоугольника** 6](#_Toc135411992)

[**Метод трапеции** 7](#_Toc135411993)

[**Метод Симпсона** 7](#_Toc135411994)

[**Метод Симпсона для решения двойного интеграла** 7](#_Toc135411995)

[**Решение системы дифф. ур. разными способами** 8](#_Toc135411996)

[**Метод Эйлера** 8](#_Toc135411997)

[**Метод Рунга-Кутта 2-го порядка** 9](#_Toc135411998)

[**Метод Рунга-Кутта 4-го порядка** 9](#_Toc135411999)

[**Метод Адамса** 10](#_Toc135412000)

[**Список литературы** 11](#_Toc135412001)

# **Поиск корня уравнения скалярной функции**

class MethodSearchRoot

public delegate double Func(double x);

## **Метод половинного деления**

public static double HalfDivision(double a, double b, double eps, Func f)

Метод половинного деления (HalfDivision) используется для нахождения корня уравнения f(x) = 0 на заданном интервале [a, b]. Он работает следующим образом:

1. Вычисляем значения функции f в точках a и b: fa = f(a), fb = f(b).
2. Проверяем знаки fa и fb. Если fa \* fb > 0, то корня на интервале [a, b] нет и возвращается значение NaN (Not-a-Number).
3. Иначе начинаем итерационный процесс: на каждой итерации находим середину интервала c = (a + b) / 2 и вычисляем значение функции fc = f(c).
4. Если fa \* fc < 0, то корень находится на интервале [a, c], и мы заменяем b на c. Иначе корень находится на интервале [c, b], и мы заменяем a на c, а fa на fc.
5. Повторяем шаги 3-4 до тех пор, пока длина текущего интервала [a, b] не станет меньше заданной точности eps.
6. Возвращаем середину этого интервала как найденное значение корня уравнения.

## **Метод Ньютона**

public static double MethodNewton(double t, double eps, Func f)

Метод Ньютона (MethodNewton) также используется для нахождения корня уравнения f(x) = 0, но он не требует задания начального интервала, а использует лишь одну начальную точку t. Он работает следующим образом:

1. Вычисляем значение функции f и ее производной f' в точке t: ft = f(t), fpt = (f(t + eps) - ft) / eps.
2. Находим приближенное значение корня уравнения, используя формулу x = t - ft / fpt.
3. Вычисляем текущую погрешность delta = |x - t|.
4. Проверяем, что delta меньше заданной точности eps. Если это так, то возвращаем найденное значение x.
5. Иначе обновляем значение t = x и переходим к шагу 1.
6. Если при следующем вычислении delta оказалось больше, чем на предыдущей итерации, то возвращается значение NaN, так как метод может не сойтись.

## **Метод последовательного приближения**

public static double SequentialApproximation(double t, double eps, Func f)

Метод последовательного приближения (SequentialApproximation) также используется для нахождения корня уравнения x = f(x) на основе одной начальной точки t. Он работает следующим образом:

1. Вычисляем значение функции f в точке t: ft = f(t).
2. Находим новую точку x = f(t).
3. Вычисляем текущую погрешность delta = |x - t|.
4. Проверяем, что delta меньше заданной точности eps. Если это так, то возвращаем найденное значение x.
5. Иначе обновляем t = x

# **Матричные методы**

class Matrix

## **Методы – верхний и нижний треугольники**

public static Vector? SolveLU\_DownTriangle(Matrix A, Vector B)

public static Vector? SolveLU\_UpTriangle(Matrix A, Vector B)

Методы SolveLU\_DownTriangle и SolveLU\_UpTriangle решают систему линейных уравнений для треугольных матриц (нижней и верхней соответственно) с использованием метода прямого хода и метода обратного хода соответственно. Они принимают на вход матрицу A и вектор B, и возвращают вектор решения x. Перед выполнением вычислений методы проверяют соответствие размерностей матрицы и вектора. Если размерности не совпадают, методы выводят сообщение об ошибке и возвращают значение null. Также методы проверяют, что все элементы на главной диагонали матрицы не равны нулю, чтобы избежать деления на ноль. Если какой-либо элемент на главной диагонали равен нулю, методы выводят сообщение об ошибке и возвращают значение null.

## **Метод Гаусса**

public static Vector MethodGauss(Matrix AA, Vector bb)

Метод MethodGauss решает систему линейных уравнений методом Гаусса. Он принимает на вход матрицу A и вектор B, и возвращает вектор решения x. Перед выполнением вычислений метод проверяет соответствие размерностей матрицы и вектора. Если размерности не совпадают, метод выводит сообщение об ошибке и возвращает значение null. *-----Путем-----* Метод использует прямой ход метода Гаусса для приведения матрицы к треугольному виду и обратный ход для нахождения решения системы. Если в процессе выполнения алгоритма метод обнаруживает нулевой элемент на главной диагонали, он выбрасывает исключение с сообщением об ошибке.

В целом, все методы выполняют примерно одинаковые действия: они проверяют входные данные, приводят матрицу к треугольному виду и находят решение системы. Разница между ними заключается в том, что методы SolveLU\_DownTriangle и SolveLU\_UpTriangle работают только с треугольными матрицами, тогда как метод MethodGauss работает с произвольными матрицами.

## **Метод квадратных корней**

public static Vector? MethodSquareRoot(Matrix aa, Vector bb)

Метод квадратных корней - это численный метод решения систем линейных уравнений, основанный на факторизации матрицы А в произведение нижнетреугольной матрицы L и ее транспонированной матрицы.

Алгоритм метода заключается в следующем:

1. Сначала находим нижнетреугольную матрицу L путем применения алгоритма Холецкого к матрице A. В начале определяется первый элемент матрицы L, который равен квадратному корню из первого элемента матрицы A. Затем последовательно вычисляются оставшиеся элементы матрицы L, используя уже вычисленные значения элементов.
2. Затем транспонируем матрицу L и получаем верхнетреугольную матрицу LT.
3. Решаем систему Ly = b, находим вектор y, где L - нижнетреугольная матрица.
4. Решаем систему LTx = y, находим вектор x, где LT - транспонированная матрица L.

Метод квадратных корней используется для решения систем линейных уравнений с симметрической и положительно определенной матрицей А. При выполнении этих условий метод квадратных корней является более эффективным, чем метод Гаусса или метод прогонки.

## **Метод Грамма-Шмидта**

public static Vector MethodGramSchmidt(Matrix A, Vector B)

Метод Грамма-Шмидта решает систему линейных уравнений Ax=B, где A - квадратная матрица, B - вектор-столбец правой части системы.

В методе сначала производится проверка возможности решения системы, инициализация необходимых переменных и матриц. Затем происходит построение ортонормированной системы векторов, на основе метода Грамма-Шмидта. В процессе этого вычисляются матрицы R и T, а также вектор y. *Что за T????*

В конце метода вызывается метод SolveLU\_UpTriangle, который решает систему с треугольной матрицей, чтобы получить вектор x, являющийся решением исходной системы линейных уравнений.

## **Метод последовательных приближений**

public static Vector MethodSuccessiveApproximations(Matrix aa, Vector bb)

Метод последовательных приближений решает систему линейных уравнений Ax=B, где A - квадратная матрица, B - вектор-столбец правой части системы. Существуют разные способы привидения.

Сначала система приводится к виду x = aphfa \* x + beta

В методе также производится проверка возможности решения системы и инициализация необходимых переменных. Затем происходит построение матрицы alpha и вектора beta. Вектор beta является первым приближением решения системы. Затем в цикле вычисляются последующие приближения решения, пока разница между текущим и предыдущим векторами решения не будет меньше заданного значения eps.

Для вычисления каждого приближения решения метод использует формулу xCurrent = beta - alpha \* xPrev, где xPrev - предыдущее приближение решения.

# **Сплайн функции(интерполяция)**

class SplineInterpolation

public SplineInterpolation(Vector x, Vector y)

public double GetInterpolation(double xValue)

Кубическая интерполяция - это метод аппроксимации функции, который используется для оценки значения функции в точках между известными значениями. Она основана на предположении, что функция между двумя соседними точками может быть аппроксимирована кубическим полиномом.

Идея кубической интерполяции заключается в том, чтобы построить кубический полином, который проходит через две соседние точки и имеет такую же первую и вторую производные в этих точках, как и исходная функция. Это позволяет достичь более гладкой и непрерывной аппроксимации функции.

Для построения кубической интерполяции обычно используется метод сплайнов. Сплайн - это гладкий полином, который состоит из нескольких сегментов, каждый из которых представляет собой кубическую функцию. Кубическая интерполяция с использованием сплайнов обеспечивает плавный переход между соседними сегментами и минимизирует искажения, которые могут возникнуть при аппроксимации.

Основное преимущество кубической интерполяции заключается в том, что она позволяет достаточно точно оценить значения функции в промежуточных точках на основе ограниченного количества известных точек. Это делает ее полезным инструментом для интерполяции и аппроксимации данных в различных областях, таких как численное моделирование, компьютерная графика и анализ данных.

Класс SplineInterpolation представляет собой реализацию кубического сплайна для интерполяции функции, заданной таблично. Кубический сплайн является кусочно-полиномиальной функцией третьей степени, которая аппроксимирует исходную функцию на каждом отрезке между узлами сетки.Входные данные для работы класса - это два массива (вектора) x и y одинаковой длины, которые содержат координаты точек, через которые должна проходить интерполированная кривая.

Конструктор класса SplineInterpolation принимает два аргумента: вектор x, содержащий узлы сетки, и вектор y, содержащий значения функции в этих узлах. При создании объекта класса происходит инициализация внутренних полей объекта (n, x, y, m, h) и вычисление коэффициентов кубического сплайна с помощью метода Coefficients.

Метод Coefficients() вычисляет коэффициенты интерполяции для каждого сегмента. Внутри этого метода вычисляются векторы alpha и beta, а затем вычисляются значения m - коэффициенты сплайнов.

Метод GetInterpolation() производит интерполяцию в точке xValue. Сначала определяется сегмент, в котором находится xValue. Затем вычисляются коэффициенты a, b, и c для этого сегмента, и на их основе производится интерполяция.

Метод FindSegment() используется в GetInterpolation() для определения сегмента, в котором находится xValue.

SplineInterpolation возвращает результаты интерполяции - значение функции в точке xValue.

В итоге, класс SplineInterpolation позволяет выполнить интерполяцию функции с использованием кубического сплайна, что может быть полезно в различных задачах анализа данных и моделирования.

# **Метод наименьших квадратов (МНК)**

class LeastSquares

public delegate double FuncPsi(double func);

public FuncPsi[] func;

public LeastSquares(Vector x, Vector y, FuncPsi[] func)

public double GetCriteria()

МНК (Метод наименьших квадратов) - это статистический метод, используемый для оценки параметров математической модели путем минимизации суммы квадратов отклонений между наблюдаемыми значениями и значениями, предсказанными моделью. Идея МНК состоит в том, чтобы найти линию или кривую, которая наилучшим образом соответствует имеющимся данным. Он широко применяется в различных областях, включая экономику, физику, инженерию и социальные науки, для анализа и прогнозирования данных.

Данный класс представляет собой реализацию метода наименьших квадратов (МНК) для аппроксимации функции. Конструктор принимает векторы x и y, представляющие собой соответствующие значения аргумента и функции соответственно, а также массив пси-функций func, которые используются для аппроксимации.

Класс содержит также вектор параметров p, который является результатом вычисления методом МНК, а также количество аргументов n и количество пси-функций m.

Данный код реализует метод наименьших квадратов для аппроксимации функции, заданной набором точек. Входными данными являются векторы x и y, содержащие соответствующие аргументы и значения функции, а также массив func, содержащий пси-функции. Методы класса Least Squares позволяют получить вектор параметров p, который является результатом аппроксимации, а также значение критерия МНК, выражающего отклонение аппроксимирующей функции от исходных данных.

Метод Parameters() используется для вычисления вектора параметров p. Он создает матрицу H, заполняет ее пси-функциями для каждого значения аргумента и вычисляет вектор параметров p с использованием формулы МНК.

Общая формула МНК выглядит следующим образом: p = (H^T \* H)^(-1) \* H^T \* y, где:

* p - вектор параметров модели;
* H - матрица пси-функций, каждая строка которой соответствует одному наблюдению, а каждый столбец - одной пси-функции;
* H^T - транспонированная матрица H;
* y - вектор значений, которые требуется аппроксимировать;
* ^(-1) - оператор обратной матрицы.

Метод GetFunc() используется для вычисления вектора пси-функций для заданного значения аргумента x.

Метод GetCriteria() вычисляет критерий МНК, который является мерой отклонения реальных значений функции от аппроксимации с использованием вычисленных параметров. Он создает вектор result, который содержит разность между реальными значениями функции и аппроксимацией с использованием вычисленных параметров. Затем он вычисляет норму L1 этого вектора и возвращает ее в качестве критерия МНК.

Общее назначение класса LeastSquares заключается в том, чтобы предоставить удобный способ вычисления параметров пси-функций для аппроксимации функции с использованием метода наименьших квадратов.

# **Вычисление определенного интеграла разными способами**

public delegate double Integral(double x);

public delegate double DoubleIntegral(double x, double y);

class Integrals

## **Метод прямоугольника**

public static double MethodRectangle(double a, double b, double eps, Integral f)

Метод прямоугольников (MethodRectangle): Метод прямоугольников, также известный как метод левых прямоугольников, заключается в том, чтобы разбить отрезок интегрирования на n равных частей и заменить подынтегральную функцию f(x) на f(xᵢ₋₁), где xᵢ = a + i\*h, а h - шаг разбиения. Значение интеграла приближенно вычисляется как сумма площадей прямоугольников шириной h и высотой f(xᵢ₋₁). Погрешность метода оценивается как O(h²).

## **Метод трапеции**

public static double MethodTrapezoid(double a, double b, double eps, Integral f)

Метод трапеций (MethodTrapezoid): Метод трапеций заключается в том, чтобы разбить отрезок интегрирования на n равных частей и заменить подынтегральную функцию на линейную интерполяцию между f(xᵢ₋₁) и f(xᵢ). Значение интеграла приближенно вычисляется как сумма площадей трапеций высотой h и основаниями f(xᵢ₋₁) и f(xᵢ). Погрешность метода оценивается как O(h²).

Формула метода трапеции для вычисления определенного интеграла имеет следующий вид:

∫[a, b] f(x) dx ≈ Δx \* [f(a)/2 + f(x₁) + f(x₂) + ... + f(b)/2], где:

* [a, b] - интервал интегрирования,
* f(x) - интегрируемая функция,
* Δx = (b - a)/n, где n - количество интервалов или "шагов" разбиения,
* x₁, x₂, ..., xn-1 - значения, соответствующие равномерно распределенным точкам на интервале [a, b].

## **Метод Симпсона**

public static double MethodSimpsons(double a, double b, double eps, Integral f)

Метод Симпсона (MethodSimpsons): Метод Симпсона заключается в том, чтобы разбить отрезок интегрирования на четное число n точек, заменить подынтегральную функцию на квадратичную интерполяцию между f(xᵢ₋₁), f(xᵢ) и f(xᵢ₊₁). Значение интеграла приближенно вычисляется как сумма площадей парабол высотой h и вершинами f(xᵢ₋₁), f(xᵢ) и f(xᵢ₊₁). Погрешность метода оценивается как O(h⁴).

Формула метода Симпсона для вычисления определенного интеграла функции f(x) на отрезке [a, b] состоит из следующих шагов:

1. Разбейте интервал [a, b] на четное число подинтервалов равной ширины h. Для этого можно выбрать равномерную сетку с шагом h = (b - a) / n, где n - четное число.
2. Вычислите значения функции f(x) в узлах сетки: f(x0), f(x1), ..., f(xn), где xi = a + i \* h для i = 0, 1, ..., n.
3. Примените формулу Симпсона для каждой пары подинтервалов:

∫[xi, xi+2] f(x) dx ≈ (h/3) \* [f(xi) + 4 \* f(xi+1) + f(xi+2)]

Здесь xi и xi+2 - границы подинтервала, а xi+1 - середина подинтервала.

1. Суммируйте результаты для всех пар подинтервалов:

∫[a, b] f(x) dx ≈ (h/3) \* [f(x0) + 4 \* f(x1) + 2 \* f(x2) + 4 \* f(x3) + ... + 2 \* f(xn-2) + 4 \* f(xn-1) + f(xn)]

Здесь x0 = a, xn = b.

## **Метод Симпсона для решения двойного интеграла**

public static double MethodSimpsonDouble(double a, double b, double c, double d, double eps, DoubleIntegral f)

Аргументы метода:

* a, b, c, d - границы области интегрирования;
* eps - точность вычисления;
* f - функция, которую нужно проинтегрировать.

Метод начинает с небольшого числа точек (2 по каждой оси) и на каждой итерации удваивает их количество, чтобы улучшить точность результата.

На каждой итерации метод разбивает прямоугольную область интегрирования на более мелкие подобласти с помощью шагов hx и hy по соответствующим осям. Затем на каждой подобласти вычисляется значение функции f(x,y) в узлах с помощью вложенных циклов for. Внутренний цикл перебирает узлы по оси y, а внешний - по оси x.

Далее метод вычисляет значение интеграла на текущей итерации с помощью формулы Симпсона. Формула применяется на каждой подобласти с весами, зависящими от положения узла на оси x и y. Затем все значения интегралов на подобластях суммируются.

Если разность между значением интеграла на текущей итерации и предыдущей не превосходит заданную точность eps, то метод останавливается и возвращает результат. Иначе он удваивает количество точек по обеим осям и повторяет процесс.

В итоге метод возвращает численное значение двойного интеграла функции f(x,y) на прямоугольной области [(a,b), (c,d)].

Формула метода Симпсона для вычисления двойного интеграла функции f(x, y) на прямоугольной области [(a, b), (c, d)] имеет вид:

∫∫[(a,b),(c,d)] f(x, y) dxdy ≈ (hx \* hy / 9) \* ΣΣ [w(i,j) \* f(x\_i, y\_j)], где:

* hx = (b - a) / nx - шаг по оси x, разбиение на nx частей,
* hy = (d - c) / ny - шаг по оси y, разбиение на ny частей,
* w(i,j) - весовой коэффициент для узла (x\_i, y\_j) в сетке,
* f(x\_i, y\_j) - значение функции f(x, y) в узле (x\_i, y\_j),
* ΣΣ - двойная сумма, перебирающая узлы сетки.

# **Решение системы дифф. ур. разными способами**

class DifferentialEquations

public delegate Vector Derivative(double t, Vector x)

public static Vector PravilaDU(double t, Vector x)

Общий вид дифференциального уравнения можно записать как: dy/dx = f(x, y)

Здесь y - неизвестная функция, зависящая от переменной x, а f(x, y) - заданная функция, определяющая связь между производной dy/dx и переменными x и y.

Дифференциальное уравнение решает задачу нахождения функции y(x), удовлетворяющей данному уравнению. Такое решение позволяет определить поведение и свойства системы, описываемой уравнением, в зависимости от входных данных и начальных условий. Дифференциальные уравнения широко используются для моделирования и анализа различных процессов и явлений в физике, инженерии, экономике, биологии и других науках.

## **Метод Эйлера**

public static Matrix MethodEuler(double t0, double tEnd, Vector x0, int steps, Derivative f)

Аргументы метода:

* t0: начальное значение независимой переменной t,
* tEnd: конечное значение независимой переменной t,
* x0: вектор начальных значений зависимых переменных,
* steps: количество шагов (или точек) для разбиения интервала [t0, tEnd],
* f: функция, описывающая систему ОДУ и возвращающая вектор производных.

Метод Эйлера представляет собой простой численный метод, который использует локальное линейное приближение для численного решения системы ОДУ. Он основан на приближении изменения переменных за один шаг по формуле:

x\_i+1 = x\_i + f(t\_i, x\_i) \* dt, где:

* x\_i - значения переменных на предыдущем шаге,
* x\_i+1 - значения переменных на текущем шаге,
* t\_i - значение независимой переменной на предыдущем шаге,
* f(t\_i, x\_i) - вектор производных переменных на предыдущем шаге,
* dt - шаг по независимой переменной t.

Код начинает с инициализации необходимых переменных и создания матрицы result, которая будет содержать значения переменных x\_i на каждом шаге. Значения начальных условий x0 и t0 записываются в первый столбец матрицы result.

Затем в цикле происходит итерация от k=1 до steps. На каждой итерации:

1. Вычисляется вектор производных pr = f(t, xt) с использованием функции f.
2. Вычисляется новое значение переменных xt на текущем шаге, используя формулу метода Эйлера: xt = xt + pr \* dt.
3. Значение независимой переменной t обновляется на шаг dt: t += dt.
4. Значения t и xt записываются в столбец column, который затем помещается в матрицу result в соответствующий столбец k.

После завершения цикла, матрица result содержит значения переменных x\_i на каждом шаге. Она возвращается в качестве результата выполнения метода.

Таким образом, метод Эйлера использует локальное линейное приближение для численного решения системы ОДУ с пошаговым обновлением значений переменных на каждом шаге.

## **Метод Рунга-Кутта 2-го порядка**

public static Matrix MethodRungeKutta2(double t0, double tEnd, Vector x0, int steps, Derivative f)

Аргументы метода:

* t0: начальное значение независимой переменной t,
* tEnd: конечное значение независимой переменной t,
* x0: вектор начальных значений зависимых переменных x,
* steps: количество шагов интегрирования,
* f: функция, возвращающая производную системы ОДУ в виде вектора.

Метод начинает с инициализации матрицы result размером (n + 1) x (steps + 1), где n - размерность вектора x0. Каждый столбец матрицы result будет содержать значения t и соответствующие значения x на текущем шаге.

Затем вычисляется шаг интегрирования dt, определяющий разность между значениями t на соседних шагах.

Далее создается вектор column размером (n + 1) для хранения значений t и x на текущем шаге. В первый элемент column записывается значение t0, а в остальные элементы записываются начальные значения x из вектора x0. Затем column добавляется в первый столбец матрицы result.

Далее происходит цикл по шагам интегрирования от 1 до steps. На каждом шаге выполняются следующие действия:

1. Вычисляются значения производной k1 = f(t, x) в текущей точке.
2. Вычисляются значения производной k2 = f(t + dt, x + k1 \* dt) в следующей точке, используя значения k1 и dt.
3. Обновляются значения переменной x, используя полученные значения производных k1 и k2 по формуле x = x + (k1 + k2) \* 0.5 \* dt.
4. Обновляется значение переменной t путем прибавления шага интегрирования dt.
5. Обновляется вектор column с новыми значениями t и x.
6. Обновленный вектор column добавляется в соответствующий столбец матрицы result.

По завершении цикла метод возвращает матрицу result, содержащую значения t и x на каждом шаге интегрирования.

## **Метод Рунга-Кутта 4-го порядка**

public static Matrix MethodRungeKutta4(double t0, double tEnd, Vector x0, int steps, Derivative f)

Аргументы метода:

* t0, tEnd - начальное и конечное значения времени,
* x0 - вектор начальных условий,
* steps - количество шагов интегрирования,
* f - функция, представляющая систему ОДУ и возвращающая производные по времени.

Метод Рунге-Кутты 4-го порядка представляет собой итерационный процесс, который вычисляет приближенное численное решение системы ОДУ на каждом шаге. Он использует комбинацию четырех коэффициентов k1, k2, k3 и k4, которые вычисляются в каждой итерации.

В начале метод создает матрицу result, в которую записываются значения времени и компоненты вектора x на каждом шаге интегрирования. Затем задается шаг по времени dt, инициализируется вспомогательный вектор column, в котором хранятся значения времени и компоненты x, и записывается начальное состояние в матрицу result.

Далее следует основной цикл метода, который выполняется на каждом шаге интегрирования. На каждой итерации вычисляются значения производных k1, k2, k3 и k4 с использованием функции f и текущих значений времени и компонент x. Затем происходит обновление значений вектора x с помощью взвешенной комбинации этих производных и шага времени dt.

После обновления значений вектора x и времени, они записываются в вспомогательный вектор column. Затем column добавляется как новая колонка в матрицу result.

По завершении всех шагов интегрирования метод возвращает матрицу result, содержащую значения времени и решения системы ОДУ на каждом шаге.

Формула для обновления значений вектора x на каждой итерации метода Рунге-Кутты 4-го порядка имеет вид: x = x + (k1 + 2 \* k2 + 2 \* k3 + k4) \* dt / 6.0, где:

* x - вектор компонент решения,
* k1, k2, k3, k4 - промежуточные коэффициенты,
* dt - шаг по времени.

## **Метод Адамса**

public static Matrix MethodAdams4(double t0, double tEnd, Vector x0, int steps, Derivative f)

Метод Адамса является численным методом для решения обыкновенных дифференциальных уравнений (ОДУ). Его идея состоит в приближенном нахождении значений функции на последовательных временных шагах.

Метод Адамса 4-го порядка отличается от 2-го и 3-го порядка своей точностью аппроксимации. Порядок аппроксимации определяет, насколько точно численный метод приближает реальное решение. Чем выше порядок, тем более точное приближение может быть получено.

Метод Адамса 4-го порядка использует комбинацию предыдущих значений функции и их производных для приближенного вычисления значения на следующем временном шаге. Он использует 4 предыдущие точки для вычисления следующей. Это позволяет достичь четвёртого порядка точности аппроксимации.

В отличие от этого, метод Адамса 2-го порядка использует только 2 предыдущие точки, а метод Адамса 3-го порядка использует 3 предыдущие точки. Соответственно, метод Адамса 4-го порядка обеспечивает более точное приближение и обладает более высокой точностью аппроксимации, чем методы второго и третьего порядков.

Алгоритм работы метода:

1. Вычисление шага h на основе заданного временного интервала (t0 - tEnd) и числа шагов steps.
2. Создание матрицы result размером (steps + 1) x (размерность вектора x0), которая будет содержать результаты численного решения.
3. Инициализация первых 4 шагов, используя метод Рунге-Кутты. На каждом шаге происходит вычисление вектора xi, используя четыре итерации метода Рунге-Кутты с соответствующими коэффициентами. Результаты сохраняются в матрице result.
4. Применение метода Адамса для остальных шагов (от 4 до steps). На каждом шаге происходит два этапа:
   * Предсказание (predictor): Используя значения xi и предыдущих шагов, вычисляется предсказанное значение xi+1 с помощью формулы Адамса.
   * Коррекция (corrector): Используя предсказанное значение predictor и значения на предыдущих шагах, вычисляется корректированное значение xi+1 с помощью формулы Адамса.

Результаты сохраняются в матрице result.

1. Создание вектора time, содержащего значения времени на каждом шаге.
2. Добавление столбца со значениями времени в матрицу result.
3. Транспонирование матрицы result (для удобства представления результатов).
4. Возвращение результирующей матрицы, содержащей численное решение СДУ на заданном временном интервале.

Формула Адамса (Адамса-Башфорта) используется для численного решения обыкновенного дифференциального уравнения (ОДУ) с использованием предыдущих значений функции и её производных.

Формула Адамса для предсказания значения y(x) на следующем шаге x + h выглядит следующим образом:

y\_predictor(x + h) = y(x) + h \* [B₀ \* f(x, y(x)) + B₁ \* f(x - h, y(x - h)) + B₂ \* f(x - 2h, y(x - 2h)) + ... + Bₖ \* f(x - kh, y(x - kh))], где:

y(x) - значение функции на текущем шаге x,

f(x, y) - производная функции, определенная дифференциальным уравнением,

h - шаг интегрирования,

B₀, B₁, B₂, ..., Bₖ - коэффициенты формулы Адамса, которые зависят от порядка метода.

Для метода Адамса-Башфорта 4-го порядка (как в приведенном коде) коэффициенты имеют следующие значения:

B₀ = 55/24,

B₁ = -59/24,

B₂ = 37/24,

B₃ = -9/24.

Таким образом, формула Адамса для предсказания значения y(x + h) будет:

y\_predictor(x + h) = y(x) + h \* [(55/24) \* f(x, y(x)) - (59/24) \* f(x - h, y(x - h)) + (37/24) \* f(x - 2h, y(x - 2h)) - (9/24) \* f(x - 3h, y(x - 3h))].

Затем после предсказания, используется коррекция для получения более точного значения на следующем шаге.

Обратите внимание, что формула Адамса может использоваться для численного решения ОДУ различных порядков, и коэффициенты B₀, B₁, B₂, ... соответственно меняются в зависимости от порядка метода.

# **Список литературы**

1. DemidovichMaron1966ru
2. Калиткин Н.Н. Численные методы
3. Исаков В.Б. Элементы численных методов
4. Воеводин В.В., Кузнецов Ю.А. Матрицы и вычисления