|  |
| --- |
| Московский Авиационный Институт |
| (Национальный Исследовательский Университет) |
| Факультет информационных технологий и прикладной математики |
| Лабораторные работы по курсу «Численные методы» |
| |  |  | | --- | --- | | Выполнил: | Козырев П. А. | | Преподаватель: | Ревизников Д.Л. | | Вариант: | 13 | | Оценка: |  | |
| Москва, 2025 |

Оглавление

[Лабораторная работа №1 3](#_Toc198150473)

[Алгоритм LU-разложения матрицы 3](#_Toc198150474)

[Метод прогонки 6](#_Toc198150475)

[Итерационные методы решения СЛАУ 9](#_Toc198150476)

[Метод вращений 14](#_Toc198150477)

[QR-алгоритм нахождения собственных значений матриц 16](#_Toc198150478)

[Лабораторная работа №2 20](#_Toc198150479)

[Решение нелинейных уравнений 20](#_Toc198150480)

[Решение систем нелинейных уравнений 23](#_Toc198150481)

[Лабораторная работа №3 26](#_Toc198150482)

[Интерполяция 26](#_Toc198150483)

[Сплайн 29](#_Toc198150484)

[Метод наименьших квадратов 32](#_Toc198150485)

[Численное дифференцирование 34](#_Toc198150486)

[Численное интегрирование 35](#_Toc198150487)

[Лабораторная работа №4 39](#_Toc198150488)

[Численные методы решения задачи Коши 39](#_Toc198150489)

[Численные методы решения краевой задачи для ОДУ 43](#_Toc198150490)

# **Лабораторная работа №1**

## **Алгоритм LU-разложения матрицы**

‑разложение матрицы представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т. е. , где — нижняя треугольная матрица, — верхняя треугольная матрица.

‑разложение может быть построено с использованием метода Гаусса. Рассмотрим -й шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов -го столбца матрицы . С этой целью используется операция:

В терминах матричных операций такая операция эквивалентна умножению , где элементы матрицы определяются следующим образом

В результате прямого хода метода Гаусса получим ,

де — верхняя треугольная матрица, а — нижняя треугольная матрица, имеющая вид

В дальнейшем -разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида . Действительно, подставляя -разложение в СЛАУ, получим , или . Т.е. процесс решения СЛАУ сводится к двум простым этапам:

1. На первом этапе решается СЛАУ . Поскольку матрица системы – нижняя треугольная, решение можно записать в явном виде:
2. На втором этапе решается СЛАУ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

Зная -разложение легко также получить определитель матрицы и обратную ей матрицу. Определитель находится как произведение элементов на главных диагоналях матриц и :

Обратную матрицу можно найти из отношения . Это уравнение также можно решить методом ‑разложения.

**Входные данные:** Изображение выглядит как рукописный текст, Шрифт, текст, белый

Контент, сгенерированный ИИ, может содержать ошибки.

**Метод для LU-разложения:**

def lu\_decomposition*(*A*)*:  
 n = len*(*A*)* LU = copy*(*A*)* for k in range*(*n*)*:  
 for i in range*(*k + 1, n*)*:  
 LU*[*i*][*k*]* /= LU*[*k*][*k*]* for j in range*(*k + 1, n*)*:  
 LU*[*i*][*j*]* -= LU*[*i*][*k*]* \* LU*[*k*][*j*]* return LU

**Функция для решения СЛАУ с помощью LU-разложения:**

def solve\_system*(*A, b*)*:  
 n = len*(*A*)* LU = lu\_decomposition*(*A*)* L = *[[*0*]* \* n for \_ in range*(*n*)]* U = *[[*0*]* \* n for \_ in range*(*n*)]* for i in range*(*n*)*:  
 for j in range*(*i + 1*)*:  
 L*[*i*][*j*]* = LU*[*i*][*j*]* for j in range*(*i, n*)*:  
 U*[*i*][*j*]* = LU*[*i*][*j*]* # 1. Ly = b  
 y = forward\_substitution*(*L, b*)* # 2.Ux = y  
 x = backward\_substitution*(*U, y*)* return x

**Результат:**

Решение системы:

[-1.999999999999999, -3.0000000000000053, -6.000000000000004, -6.999999999999996]

Определитель:

-344.99999999999994

Обратная матрица:

-0.07246376811594203 0.07246376811594198 -0.2028985507246377 0.14492753623188406

0.005797101449275383 -0.6057971014492752 -0.2637681159420288 -0.2115942028985507

-0.020289855072463753 -0.3797101449275363 -0.5768115942028987 0.24057971014492757

-0.06666666666666668 0.4666666666666666 0.5333333333333333 -0.0666666666666667

Проверка A\*A^-1:

1.0 0.0 0.0 -0.0

0.0 1.0 0.0 -0.0

0.0 -0.0 1.0 0.0

0.0 -0.0 -0.0 1.0

## Метод прогонки

Метод прогонки является частным случаем метода Гаусса. Он применяется для решения СЛАУ с трехдиагональными матрицами. Рассмотрим СЛАУ:

При этом будем полагать, что и . Решение системы можно искать в виде:

Здесь и – прогоночные коэффициенты, определяемые по формулам:

После того как будут найдены прогоночные коэффициенты (прямой ход), можно вычислить значения неизвестных путем обратной подстановки (обратный ход):

Достаточным условием корректности метода прогонки и устойчивости его к погрешностям вычислений является условие преобладания диагональных коэффициентов:

**Входные данные:** Изображение выглядит как текст, Шрифт, рукописный текст, белый

Контент, сгенерированный ИИ, может содержать ошибки.

**Достаточное условие:**

def checking*(*arr*)*:  
 n = len*(*arr*)* for i in range*(*1, n-1*)*:  
 if arr*[*i*][*0*]* == 0 or arr*[*i*][*2*]* == 0:  
 return False  
  
 flag = 0  
 for i in range*(*n*)*:  
 if i == 0:  
 if abs*(*arr*[*i*][*0*])* == abs*(*arr*[*i*][*1*])*:  
 flag = 1  
 if abs*(*arr*[*i*][*0*])* < abs*(*arr*[*i*][*1*])*:  
 return False  
 elif i == n - 1:  
 if abs*(*arr*[*i*][*1*])* == abs*(*arr*[*i*][*0*])*:  
 flag = 1  
 if abs*(*arr*[*i*][*1*])* < abs*(*arr*[*i*][*0*])*:  
 return False  
 else:  
 if abs*(*arr*[*i*][*1*])* == *(*abs*(*arr*[*i*][*0*])* + abs*(*arr*[*i*][*2*]))*:  
 flag = 1  
 if abs*(*arr*[*i*][*1*])* < *(*abs*(*arr*[*i*][*0*])* + abs*(*arr*[*i*][*2*]))*:  
 return False  
  
 if*(*flag*)*:  
 return True  
 else:  
 return False

**Метод решения СЛАУ с трёхдиагональной матрицей:**

def method\_progonki*(*arr*)*:  
 if checking*(*arr*)*:  
 print*(*"Проверка: Достаточное условие выполнено"*)* else:  
 print*(*"Проверка: Достаточное условие НЕ выполнено"*)* n = len*(*arr*)* p = *[*0*]* \* n  
 q = *[*0*]* \* n  
  
 p*[*0*]* = *((*-1*)* \* arr*[*0*][*1*])* / arr*[*0*][*0*]* q*[*0*]* = arr*[*0*][*2*]* / arr*[*0*][*0*]* # прямой ход  
 for i in range*(*1, n*)*:  
 if i != n - 1:  
 p*[*i*]* = *((*-1*)* \* arr*[*i*][*2*])* / *(*arr*[*i*][*1*]* + arr*[*i*][*0*]* \* p*[*i - 1*])* q*[*i*]* = *(*arr*[*i*][*3*]*-arr*[*i*][*0*]*\*q*[*i - 1*])*/*(*arr*[*i*][*1*]* + arr*[*i*][*0*]* \* p*[*i - 1*])* else:  
 p*[*i*]* = 0  
 q*[*i*]* = *(*arr*[*i*][*2*]*-arr*[*i*][*0*]* \* q*[*i - 1*])* / *(*arr*[*i*][*1*]* + arr*[*i*][*0*]* \* p*[*i - 1*])* # обратный ход  
 x = *[*0*]* \* n  
 x*[*n - 1*]* = q*[*n - 1*]* for i in range*(*n - 2, -1, -1*)*:  
 x*[*i*]* = p*[*i*]* \* x*[*i + 1*]* + q*[*i*]* return x

**Результат:**

Решение методом прогонки:

Проверка: Достаточное условие выполнено

[7.0, 3.0, -6.999999999999999, 5.000000000000001, -9.0]

## Итерационные методы решения СЛАУ

Рассмотрим СЛАУ

с невырожденной матрицей.

Приведем СЛАУ к эквивалентному виду

или в векторно-матричной форме .

Такое приведение может быть выполнено различными способами. 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 качестве нулевого приближения вектора неизвестных примем вектор правых частей . Тогда **метод простых итераций** примет вид:

Достаточным условием сходимости является диагональное преобладание матрицы по строкам или по столбцам:

Критерием окончания итерационного процесса может служить неравенство .

Метод простых итераций довольно медленно сходится. Для его ускорения существует **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:** Изображение выглядит как Шрифт, текст, рукописный текст, белый

Контент, сгенерированный ИИ, может содержать ошибки.

**Метод простых итераций**

def simple\_iterations\_solve*(*a, b, eps*)*:  
 if *(*diag\_condition\_1*(*a*)* or diag\_condition\_2*(*a*))*:  
 print*(*"Достаточное условие выполнено"*)* else:  
 print*(*"Достаточное условие НЕ выполнено"*)* eps\_k = 1  
 x = *[*b*[*i*]* / a*[*i*][*i*]* for i in range*(*len*(*a*))]* iter\_count = 0  
 print*(*f"Eps = *{*eps*}*"*)* while *(*eps < eps\_k and iter\_count < 100*)*:  
 x\_k = *[*0.0*]* \* n  
 for i in range*(*n*)*:  
 s = 0.0  
 for j in range*(*n*)*:  
 if j != i:  
 s += *(*-a*[*i*][*j*]* / a*[*i*][*i*])* \* x*[*j*]* x\_k*[*i*]* = *(*b*[*i*]* / a*[*i*][*i*])* + s  
  
 eps\_k = max*(*abs*(*x\_k*[*i*]* - x*[*i*])* for i in range*(*n*))* x = *[*i for i in x\_k*]* iter\_count += 1  
 print*(*f"iter = *{*iter\_count*}*, eps\_k = *{*eps\_k*}*, x = *{*x*}*"*)* return x, iter\_count

**Метод Зейделя**

def zeidel\_solve*(*a, b, eps*)*:  
 n = len*(*a*)* if *(*diag\_condition\_1*(*a*)* or diag\_condition\_2*(*a*))*:  
 print*(*"Достаточное условие выполнено"*)* else:  
 print*(*"Достаточное условие НЕ выполнено"*)* eps\_k = 1  
 x = *[*b*[*i*]* / a*[*i*][*i*]* for i in range*(*len*(*a*))]* iter\_count = 0  
 print*(*f"Eps = *{*eps*}*"*)* while *(*eps < eps\_k and iter\_count < 100*)*:  
  
 x\_k = *[*i for i in x*]* for i in range*(*n*)*:  
 s = 0.0  
 # Обновлённые значения для j < i  
 for j in range*(*i*)*:  
 s += a*[*i*][*j*]* \* x*[*j*]* # Старые значения для j > i  
 for j in range*(*i + 1, n*)*:  
 s += a*[*i*][*j*]* \* x\_k*[*j*]* x*[*i*]* = *(*b*[*i*]* - s*)* / a*[*i*][*i*]* eps\_k = max*(*abs*(*x*[*i*]* - x\_k*[*i*])* for i in range*(*n*))* iter\_count += 1  
 print*(*f"iter = *{*iter\_count*}*, eps\_k = *{*eps\_k*}*, x = *{*x*}*"*)* return x, iter\_count

**Результат:**

Метод простых итераций:

Достаточное условие выполнено

Eps = 1e-08

iter = 1, eps\_k = 1.922222222222222, x = [-6.640277777777778, 2.285185185185185, 6.634027777777778, -0.7888888888888888]

iter = 2, eps\_k = 0.6272762345679013, x = [-6.013001543209877, 2.2478395061728396, 6.917689043209876, -1.1079629629629633]

iter = 3, eps\_k = 0.20122256515774994, x = [-5.923438143004116, 2.0466169410150896, 6.969824781378601, -1.089027777777778]

iter = 4, eps\_k = 0.07950848765432106, x = [-5.976594632344536, 2.0009689357567444, 7.010218576746113, -1.0095192901234569]

iter = 5, eps\_k = 0.019833116914818483, x = [-5.996427749259355, 1.9920428137542552, 7.00036145657698, -1.000189693358482]

iter = 6, eps\_k = 0.006728266394752325, x = [-6.002228987665766, 1.9987710801490075, 7.001945449655192, -0.9966271696625895]

iter = 7, eps\_k = 0.0032892548620901962, x = [-6.00059633173691, 1.9995611558902322, 6.999372247038231, -0.9999164245246797]

iter = 8, eps\_k = 0.0007989498243201609, x = [-6.000246550938197, 2.0003197055759583, 7.000171196862551, -0.999780517410318]

iter = 9, eps\_k = 0.00038662779761855504, x = [-5.999914800161534, 1.9999953659900136, 6.999855265436489, -1.0001671452079366]

iter = 10, eps\_k = 0.00019811562292471052, x = [-5.999997616478842, 2.000040906669722, 7.000053381059414, -0.9999731684649729]

iter = 11, eps\_k = 7.567193719637544e-05, x = [-5.999983643967267, 1.9999813804719604, 6.999977709122217, -1.0000233212410667]

iter = 12, eps\_k = 3.483157150219185e-05, x = [-6.000005258968464, 2.0000046840155, 7.0000121316569865, -0.9999884896695646]

iter = 13, eps\_k = 1.7156831029652153e-05, x = [-5.999998530274388, 1.9999964992145052, 6.999994974825957, -1.0000038417583625]

iter = 14, eps\_k = 7.303180162843148e-06, x = [-6.000001218298383, 2.000001480520886, 7.00000227800612, -0.9999978316808886]

iter = 15, eps\_k = 3.3706570805591696e-06, x = [-5.999999549900241, 1.9999994180271874, 6.999998907349039, -1.0000009771623959]

iter = 16, eps\_k = 1.5725664592736166e-06, x = [-6.000000188990165, 2.000000309925271, 7.0000004799154985, -0.9999995915174296]

iter = 17, eps\_k = 6.998675630143225e-07, x = [-5.999999897699642, 1.9999998655749285, 6.999999780047935, -1.0000002005581856]

iter = 18, eps\_k = 3.2158115192970627e-07, x = [-6.000000042439626, 2.0000000593466023, 7.000000101629087, -0.9999999100830057]

iter = 19, eps\_k = 1.4735243247798735e-07, x = [-5.99999998073856, 1.9999999715807457, 6.999999954276655, -1.0000000401184943]

iter = 20, eps\_k = 6.664904894648771e-08, x = [-6.000000009223092, 2.0000000126554838, 7.000000020925704, -0.9999999812517566]

iter = 21, eps\_k = 3.04945917406485e-08, x = [-5.999999995945908, 1.9999999942579312, 6.999999990431112, -1.0000000084671603]

iter = 22, eps\_k = 1.3912081264777498e-08, x = [-6.000000001858392, 2.0000000026566465, 7.000000004343193, -0.9999999961570478]

iter = 23, eps\_k = 6.32685726031923e-09, x = [-5.999999999141772, 1.99999999880032, 6.999999998016336, -1.000000001765644]

Решение: [-5.999999999141772, 1.99999999880032, 6.999999998016336, -1.000000001765644]

Количество итераций: 23

Метод Зейделя:

Достаточное условие выполнено

Eps = 1e-08

iter = 1, eps\_k = 2.0293672839506174, x = [-6.640277777777778, 1.8597222222222223, 6.320949074074075, -0.8960339506172839]

iter = 2, eps\_k = 0.6075512259945128, x = [-6.171417181069959, 2.184935699588477, 6.928500300068587, -1.0758687985825333]

iter = 3, eps\_k = 0.22608494322511863, x = [-5.94533223784484, 2.014525967840268, 7.017380588578977, -1.004483281469627]

iter = 4, eps\_k = 0.046787043193647904, x = [-5.992119281038488, 1.994507025432974, 7.002298258262971, -0.9975838633441517]

iter = 5, eps\_k = 0.009502482942373547, x = [-6.001621763980862, 1.9994929446783276, 6.999444560387153, -0.9998312368550089]

iter = 6, eps\_k = 0.0013531723857340694, x = [-6.000268591595128, 2.000175458635677, 6.999919112334614, -1.000077304538561]

iter = 7, eps\_k = 0.0003191698427293943, x = [-5.999949421752398, 2.000018294407232, 7.000017331052234, -1.000006256686684]

iter = 8, eps\_k = 4.1311089005624524e-05, x = [-5.999990732841404, 1.9999944499770126, 7.000002806619388, -0.9999975363961501]

iter = 9, eps\_k = 1.0828746044744264e-05, x = [-6.000001561587449, 1.99999934936843, 6.999999461114528, -0.9999997720018058]

iter = 10, eps\_k = 1.2440059631302347e-06, x = [-6.0000003175814856, 2.0000001749465572, 6.9999999031722835, -1.0000000782403597]

iter = 11, eps\_k = 3.6550967230652986e-07, x = [-5.999999952071813, 2.0000000229279546, 7.000000016661791, -1.0000000081975415]

iter = 12, eps\_k = 3.709764406778504e-08, x = [-5.999999989169457, 1.999999994508875, 7.000000003321838, -0.9999999975244255]

iter = 13, eps\_k = 1.2291077489123836e-08, x = [-6.000000001460535, 1.999999999198531, 6.999999999487987, -0.999999999708513]

iter = 14, eps\_k = 1.09285647198476e-09, x = [-6.000000000367678, 2.0000000001715654, 6.9999999998866125, -1.0000000000780196]

Решение: [-6.000000000367678, 2.0000000001715654, 6.9999999998866125, -1.0000000000780196]

Количество итераций: 14

Process finished with exit code 0

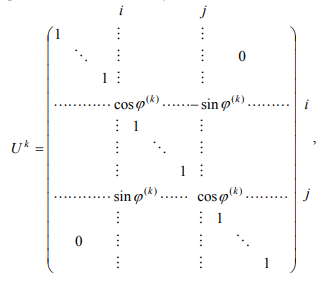
## Метод вращений

Метод вращений Якоби применим только для симметрических матриц ) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы в преобразовании подобия , а поскольку для симметрических матриц матрица преобразования подобия является ортогональной (), то , где – диагональная матрица с собственными значениями на главной диагонали.

Пусть дана симметрическая матрица . Требуется для нее вычислить с точностью все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращений:

Пусть известна матрица на k–й итерации, при этом для k=0: .

1. Выбирается максимальный по модулю недиагональный элемент матрицы .
2. 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

В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

Координатными столбцами собственных векторов матрицы в единичном базисе будут столбцы матрицы .

**Входные данные:** Изображение выглядит как текст, Шрифт, число, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

**Метод вращений:**

# Матрица поворота  
def get\_rotation\_matrix*(*n, phi, i, j*)*:  
 u = eye*(*n*)* u*[*i*][*i*]* = math.cos*(*phi*)* u*[*i*][*j*]* = -math.sin*(*phi*)* u*[*j*][*i*]* = math.sin*(*phi*)* u*[*j*][*j*]* = math.cos*(*phi*)* return u

# Координаты максимума  
def get\_coors\_of\_max*(*a*)*:  
 n = len*(*a*)* i\_max = 0  
 j\_max = 1  
 for i in range*(*n*)*:  
 for j in range*(*n*)*:  
 if i == j:  
 continue  
 if abs*(*a*[*i*][*j*])* > abs*(*a*[*i\_max*][*j\_max*])*:  
 i\_max, j\_max = i, j  
 return i\_max, j\_max  
  
# Вычисление угла phi  
def calc\_phi*(*a, i, j*)*:  
 if a*[*i*][*i*]* == a*[*j*][*j*]*:  
 return math.pi / 4  
 return 0.5 \* math.atan2*((*2 \* a*[*i*][*j*])*, *(*a*[*i*][*i*]* - a*[*j*][*j*]))*

def jacobi\_rotations*(*a, eps*)*:  
 if*(*check\_symmetry*(*a*)* == False*)*:  
 raise ValueError*(*"Матрица НЕ симметрична"*)* n = len*(*a*)* a\_k = *[*row*[*:*]* for row in a*]* u = eye*(*n*)* iter\_count = 0  
 while mat\_norm*(*a\_k*)* > eps:  
 iter\_count += 1  
 i, j = get\_coors\_of\_max*(*a\_k*)* phi = calc\_phi*(*a\_k, i, j*)* u\_k = get\_rotation\_matrix*(*n, phi, i, j*)* u = multiply\_matrices*(*u, u\_k*)* tmp = multiply\_matrices*(*transpose*(*u\_k*)*, a\_k*)* a\_k = multiply\_matrices*(*tmp, u\_k*)* print*(*f"iter = *{*iter\_count*}* mat\_norm = *{*mat\_norm*(*a\_k*)}* eps = *{*eps*}*"*)* return a\_k, u, iter\_count

**Результат:**

iter = 1 mat\_norm = 2.0 eps = 1e-05

iter = 2 mat\_norm = 0.6184883437815324 eps = 1e-05

iter = 3 mat\_norm = 0.07516752558102405 eps = 1e-05

iter = 4 mat\_norm = 0.020992777932637183 eps = 1e-05

iter = 5 mat\_norm = 0.00011095096546072031 eps = 1e-05

iter = 6 mat\_norm = 1.4606880838000418e-07 eps = 1e-05

Собственные значения:

l\_1 = -7.5340246082437785

l\_2 = 6.122561006368408

l\_3 = 8.411463601875372

Собственные векторы:

x\_1 = [-0.1217428059111089, 0.3017635107695101, -0.9455778512512599]

x\_2 = [0.27660965262858606, 0.9252374689125051, 0.25965886503848534]

x\_3 = [0.9532396284678806, -0.22994436215841663, -0.19611170548943752]

Количество итераций: 6

Проверка (Ax = lx):

[0.9172132552136486, -2.2734938511574896, 7.124006762407818] = [0.9172132956109407, -2.2734937160075255, 7.124006800337267]

[1.6935594909517178, 5.664822804716467, 1.5897773801619366] = [1.6935594731688917, 5.6648228487947065, 1.5897772420425076]

[8.01814043872192, -1.9341686327498333, -1.6495864726328024] = [8.01814043872278, -1.93416863275197, -1.6495864726261062]

Process finished with exit code 0

## QR-алгоритм нахождения собственных значений матриц

В основе -алгоритма лежит представление матрицы в виде , где Q – ортогональная матрица (), а – верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению -разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.

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

Рассмотрим случай, когда необходимо обратить в нуль все элементы какого-либо вектора кроме первого, т.е. построить матрицу Хаусхолдера такую, что

Тогда вектор определится следующим образом:

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов получить ее QR – разложение.

Процедура -разложения многократно используется в -алгоритме вычисления собственных значений. Строится следующий итерационный процесс:

– производится -разложение,

– производится перемножение матриц,

…………………..

– разложение

– перемножение

Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы в произведение ортогональной и верхней треугольной матриц, а на втором – полученные матрицы перемножаются в обратном порядке.

При отсутствии у матрицы кратных собственных значений последовательность сходится к верхней треугольной матрице (в случае, когда все собственные значения вещественны) или к верхней квазитреугольной матрице (если имеются комплексносопряженные пары собственных значений).

Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать следующее неравенство: . При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

**Входные данные:** Изображение выглядит как часы, число, диаграмма, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**QR-разложение:**

def qr\_decomposition*(*A*)*:  
 n = len*(*A*)* Q = mat\_identity*(*n*)* R = mat\_copy*(*A*)* for k in range*(*n - 1*)*:  
  
 x = *[*R*[*i*][*k*]* for i in range*(*k, n*)]* e1 = *[*1.0*]* + *[*0.0*]* \* *(*n - k - 1*)* sign = 1.0 if x*[*0*]* >= 0 else -1.0  
  
 norm\_x = vector\_norm*(*x*)* v = vector\_add*(*x, vector\_mul\_scalar*(*e1, sign \* norm\_x*))* norm\_v = vector\_norm*(*v*)* v = vector\_mul\_scalar*(*v, 1.0 / norm\_v*)* H = mat\_identity*(*n*)* for i in range*(*k, n*)*:  
 for j in range*(*k, n*)*:  
 H*[*i*][*j*]* -= 2.0 \* v*[*i - k*]* \* v*[*j - k*]* Q = mat\_mul*(*Q, H*)* R = mat\_mul*(*H, R*)* return Q, R

**QR-алгоритм:**

def qr\_algorithm*(*A, max\_iterations=100,eps = -1e9*)*:  
 Ak = mat\_copy*(*A*)* n = len*(*Ak*)* for k in range*(*max\_iterations*)*:  
 Q, R = qr\_decomposition*(*Ak*)* Ak = mat\_mul*(*R, Q*)* for i in R:  
 print*(*\*i*)* print*(*""*)* for i in range*(*n*)*:  
 sm = 0  
 for j in range*(*i,n*)*:  
 sm += A*[*i*][*j*]*\*\*2  
  
 if*(*math.sqrt*(*sm*)* < eps*)*:  
 break  
  
 for i in Ak:  
 print*(*\*i*)* return *[*Ak*[*i*][*i*]* for i in range*(*n*)]*

**Результат:**

Приближённые собственные значения:

[-13.006364910252419, 7.520358216832139, 1.4860066934203386]

# Лабораторная работа №2

## Решение нелинейных уравнений

Метод простых итераций:

Пусть требуется решить трансцендентное уравнение вида . Предположим, что корень уравнения отделен и находится на отрезке . Кроме того, функция удовлетворяет некоторым дополнительным условиям:

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

Для того чтобы построить итерационный процесс, согласно методу простой итерации уравнение заменяется эквивалентным уравнением: , причем – непрерывная функция. Выберем некоторое нулевое приближение , а затем организуем итерационный процесс по схеме:

Условие является достаточным условием сходимости итераций, если начальное приближение выбрано в некоторой окрестности корня.

Метод Ньютона:

Предположим, что на интервале требуется определить корень уравнения . Для того чтобы построить итерационный процесс согласно методу Ньютона, непрерывная функция на интервалах должна удовлетворять условиям, аналогичным условиям в методе итераций:

Правило построения итерационной последовательности получается путем замены нелинейной функции ее линейной моделью на основе формулы Тейлора. Выберем в окрестности решения уравнения две соседние точки, так что , и запишем разложение функции в ряд Тейлора:

Учитывая, что и оставляя только линейную часть разложения ряда, запишем соотношение метода Ньютона:

Используя условие сходимости метода простых итераций, получим достаточное условие сходимости метода Ньютона в форме

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

Для завершения итерационного процесса используется правило:

**Уравнение:**

**График вблизи корня:**

|  |
| --- |
|  |
| Рисунок 1 |

**Метод простых итераций:**

def simple\_iterations*(*phi, phi\_d, eps, start\_a, start\_b*)*:  
 if *(*f*(*start\_a*)* \* f*(*start\_b*)* >= 0*)*:  
 raise ValueError*(*"f(a) \* f(b) должен быть < 0"*)* q = -1  
 if *(*abs*(*phi\_d*(*start\_a*))* >= 1*)*:  
 if *(*abs*(*phi\_d*(*start\_b*))* >= 1*)*:  
 raise ValueError*(*"Достаточное условие не выполнено"*)* else:  
 q = phi\_d*(*start\_b*)* else:  
 q = phi\_d*(*start\_a*)* iter\_count = 0  
 coef = q / *(*1 - q*)* x\_k = start\_b  
 dx = 10e9  
 while eps < coef \* dx:  
 x\_k\_next = phi*(*x\_k*)* dx = abs*(*x\_k\_next - x\_k*)* x\_k = x\_k\_next  
 iter\_count += 1  
 return x\_k, iter\_count

**Метод Ньютона:**

def newton\_method*(*f, f\_d, f\_d2, eps, start\_a, start\_b*)*:  
 if *(*f*(*start\_a*)* \* f*(*start\_b*)* >= 0*)*:  
 raise ValueError*(*"f(a) \* f(b) должен быть < 0"*)* if *(*f\_d*(*start\_a*)* \* f\_d*(*start\_b*)* < 0*)*:  
 raise ValueError*(*"f`(x) не знакопостоянна"*)* if *(*f\_d2*(*start\_a*)* \* f\_d2*(*start\_b*)* < 0*)*:  
 raise ValueError*(*"f``(x) не знакопостоянна"*)* x0 = start\_a  
 if abs*(*f*(*start\_a*)* \* f\_d2*(*start\_a*))* > *(*f\_d*(*start\_a*))* \*\* 2:  
 if abs*(*f*(*start\_b*)* \* f\_d2*(*start\_b*)* > *(*f\_d*(*start\_b*))* \*\* 2*)*:  
 raise ValueError*(*"abs(f(a) \* f``(a)) должно быть <= f`(a)\*\*2 или abs(f(b) \* f``(b)) < (f`(b))\*\*2 должен быть <= (f`(b))\*\*2"*)* else:  
 x0 = start\_b  
 else:  
 if abs*(*f*(*start\_b*)* \* f\_d2*(*start\_b*)* <= *(*f\_d*(*start\_b*))* \*\* 2*)*:  
 if *(*abs*(*f*(*start\_a*))* <= abs*(*f*(*start\_b*)))*:  
 x0 = start\_a  
 else:  
 x0 = start\_b  
 else:  
 x0 = start\_a  
 iter\_count = 0  
 x\_k = x0  
 dx = 10e9  
 while eps < dx:  
 x\_k\_next = x\_k - f*(*x\_k*)* / f\_d*(*x\_k*)* dx = abs*(*x\_k\_next - x\_k*)* x\_k = x\_k\_next  
 iter\_count += 1  
 return x\_k, iter\_count

**Результат:**

Eps = 1e-08

Simple iterations | x=0.42821147285357447 | iter\_count=19

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

Newton method: | x=0.42821147091451367 | iter\_count=5

## Решение систем нелинейных уравнений

**Метод простых итераций:**

Решение системы нелинейных уравнений вида

возможно, если все функции в системе непрерывны и дифференцируемы в окрестности решения.

Для использования метода итераций система уравнений записывается в эквивалентной форме:

где – итерирующие непрерывно дифференцируемые функции.

Тогда, если известно начальное приближение , то можно построить алгоритм метода простых итераций:

Предложение формулировки достаточного условия сходимости для многомерного случая выглядит следующим образом. Метод простых итераций сходится к решению системы нелинейных уравнений, если какая-либо норма матрицы Якоби , построенная по правым частям эквивалентной системы в замкнутой области , меньше единицы на каждой итерации:

Для практических расчетов чаще всего используют матричную норму, определенную в области решения , которую обязательно проверяют в начальном приближении:

В практических вычислениях в качестве условия окончания итераций обычно используется критерий

**Метод Ньютона:**

Пусть дана система нелинейных уравнений. Все функции системы непрерывны на некотором интервале и дифференцируемы вплоть до вторых производных.

Итерационный процесс нахождения решения, который носит название метода Ньютона для систем, записывается в виде решения матричного уравнения:

где – матрица Якоби.

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

**Система:**

**Метод простых итераций**

**Метод Ньютона**

**График вблизи корня:** **Изображение выглядит как линия, диаграмма, График, круг

Контент, сгенерированный ИИ, может содержать ошибки.**

**Метод простых итераций:**

def simple\_iteration*(*x1\_init, x2\_init, phi1, phi2, dphi1\_dx1, dphi1\_dx2, dphi2\_dx1, dphi2\_dx2, epsilon,  
 max\_iter*)*:  
 x1 = x1\_init  
 x2 = x2\_init  
  
 q = max\_jacobian\_norm*(*x1, x2, dphi1\_dx1, dphi1\_dx2, dphi2\_dx1, dphi2\_dx2*)* if q >= 1:  
 print*(*"Метод не сходится, т.к. q >= 1"*)* return *(*None, None*)* for k in range*(*max\_iter*)*:  
 x1\_new = phi1*(*x1, x2*)* x2\_new = phi2*(*x1, x2*)* error = max*(*abs*(*x1\_new - x1*)*, abs*(*x2\_new - x2*))* if *(*q / *(*1 - q*))* \* error < epsilon:  
 return *(*x1\_new, x2\_new*)*, k + 1  
  
 x1, x2 = x1\_new, x2\_new  
 print*(*f"Метод не сошелся за *{*max\_iter*}* итераций"*)* return *(*None, None*)*

**Метод Ньютона:**

def newton\_method*(*x1\_init, x2\_init, f1, f2, df1\_dx1, df1\_dx2, df2\_dx1, df2\_dx2, tol=1e-4, max\_iter=100*)*:  
 x1, x2 = x1\_init, x2\_init  
 for iteration in range*(*max\_iter*)*:  
  
 f\_v = *[*f1*(*x1, x2*)*, f2*(*x1, x2*)]* J = jacobian*(*x1, x2, df1\_dx1, df1\_dx2, df2\_dx1, df2\_dx2*)* d1, d2 = solve\_linear\_system*(*J, *[*-f\_v*[*0*]*, -f\_v*[*1*]])* x1\_new = x1 + d1  
 x2\_new = x2 + d2  
  
 max\_change = max*(*abs*(*x1\_new - x1*)*, abs*(*x2\_new - x2*))* if max\_change < tol:  
 print*(*f"Сходимость достигнута за *{*iteration + 1*}* итераций."*)* return x1\_new, x2\_new  
  
 x1, x2 = x1\_new, x2\_new  
  
 print*(*"Достигнуто максимальное число итераций."*)* return x1, x2

**Результат:**

======= Метод Ньютона =======

Сходимость достигнута за 5 итераций.

Решение Ньютоном: (0.4249023331578266, 0.977171684925049)

======= Метод простых итераций =======

Решение: x1 = 0.4249023339895519, x2 = 0.9771716851719412 достигнуто за 24 итераций

# Лабораторная работа №3

## **Интерполяция**

Пусть на отрезке задано множество несовпадающих точек (интерполяционных узлов), в которых известны значения функции . Приближающая функция такая, что выполняются равенства

называется интерполяционной.

Наиболее часто в качестве приближающей функции используют многочлены степени :

Произвольный многочлен может быть записан в виде

Здесь – многочлены степени , так называемые лагранжевы многочлены влияния, которые удовлетворяют условию и, соответственно,

а интерполяционный многочлен запишется в виде

Интерполяционный многочлен, записанный в этой форме, называется **интерполяционным многочленом Лагранжа**.

Недостатком интерполяционного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют интерполяционный многочлен в форме Ньютона.

Введем понятие разделенной разности. 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 разделенные разности первого порядка:

Разделенная разность определяется соостношениями

**Интерполяционный многочлен Ньютона** может быть представлен в виде:

Отметим, что при добавлении новых узлов первые члены многочлена Ньютона остаются неизменными.

Для повышения точности интерполяции в сумму могут быть добавлены новые члены, что требует подключения дополнительных интерполяционных узлов. При этом безразлично, в каком порядке подключаются новые узлы. Этим формула Ньютона выгодно отличается от формулы Лагранжа.

**Входные данные:**

**Метод Лагранжа:**

# Интерполяционный многочлен Лагранжа  
def lagrange\_interpolation*(*X, Y, x*)*:  
 n = len*(*X*)* L = 0  
 for i in range*(*n*)*:  
 li = 1  
 for j in range*(*n*)*:  
 if j != i:  
 li \*= *(*x - X*[*j*])* / *(*X*[*i*]* - X*[*j*])* L += Y*[*i*]* \* li  
 return L

**Метод Ньютона:**  
def divided\_differences*(*X, Y*)*:  
 n = len*(*X*)* diff\_table = *[*Y*[*:*]]* for k in range*(*1, n*)*:  
 diff\_table.append*([])* for i in range*(*n - k*)*:  
 diff = *(*diff\_table*[*k - 1*][*i + 1*]* - diff\_table*[*k - 1*][*i*])* / *(*X*[*i + k*]* - X*[*i*])* diff\_table*[*k*]*.append*(*diff*)* return diff\_table  
  
  
def newton\_interpolation*(*X, Y, x*)*:  
 n = len*(*X*)* diff\_table = divided\_differences*(*X, Y*)* result = diff\_table*[*0*][*0*]* product\_term = 1  
 for i in range*(*1, n*)*:  
 product\_term \*= *(*x - X*[*i - 1*])* result += diff\_table*[*i*][*0*]* \* product\_term  
 return result

**Результат:**

f(x\*) = 1.5403023058681398

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

a)

Результат интерполяции методом Лагранжа в точке x\* = 1.0: 1.53994923644755

Погрешность интерполяции методом Лагранжа: 0.00035306942058976887

b)

Результат интерполяции методом Лагранжа в точке x\* = 1.0: 1.541999320218936

Погрешность интерполяции методом Лагранжа: 0.0016970143507961666

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

a)

Результат интерполяции методом Ньютона в точке x\* = 1.0: 1.53994923644755

Погрешность интерполяции методом Ньютона: 0.00035306942058976887

b)

Результат интерполяции методом Ньютона в точке x\* = 1.0: 1.541999320218936

Погрешность интерполяции методом Ньютона: 0.0016970143507961666

## Сплайн

Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении сплайн-интерполяции. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени:

который в узлах интерполяции принимает значения аппрокcимируемой функции и непрерывен вместе со своими производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства производных соответствующих многочленов. На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить неизвестных . Эти коэффициенты ищутся из условий в узлах сетки.

**Входные данные: Построение сплайна:**

h = *[*0*]*for i in range*(*1, len*(*X\_i*))*:  
 h.append*(*X\_i*[*i*]* - X\_i*[*i - 1*])*A = *[[*2 \* *(*h*[*1*]* + h*[*2*])*, h*[*2*]*, 0*]]*\_d = *[*3 \* *((*f\_i*[*2*]* - f\_i*[*1*])* / h*[*2*]* - *(*f\_i*[*1*]* - f\_i*[*0*])* / h*[*1*])]*for i in range*(*3, n*)*:  
 next\_row\_A = *[*0*]* \* len*(*A*[*0*])* next\_row\_A*[*i - 3*]* = h*[*i - 1*]* next\_row\_A*[*i - 2*]* = 2 \* *(*h*[*i - 1*]* + h*[*i*])* next\_row\_A*[*i - 1*]* = h*[*i*]* A.append*(*next\_row\_A*)* \_d.append*(*3 \* *((*f\_i*[*i*]* - f\_i*[*i - 1*])* / h*[*i*]* - *(*f\_i*[*i - 1*]* - f\_i*[*i - 2*])* / h*[*i - 1*]))*next\_row\_A = *[*0*]* \* len*(*A*[*0*])*next\_row\_A*[*-1*]* = 2 \* *(*h*[*n - 1*]* + h*[*n*])*next\_row\_A*[*-2*]* = h*[*n - 1*]*A.append*(*next\_row\_A*)*\_d.append*(*3 \* *((*f\_i*[*n*]* - f\_i*[*n - 1*])* / h*[*n*]* - *(*f\_i*[*n - 1*]* - f\_i*[*n - 2*])* / h*[*n - 1*]))*def TDMAsolver*(*A, \_d*)*:  
 n = len*(*\_d*)* a = *[]* b = *[]* c = *[]* d = \_d  
 ACopy = A  
  
 a.append*(*0*)* b.append*(*ACopy*[*0*][*0*])* c.append*(*ACopy*[*0*][*1*])* for i in range*(*1, n - 1*)*:  
 a.append*(*ACopy*[*i*][*i - 1*])* b.append*(*ACopy*[*i*][*i*])* c.append*(*ACopy*[*i*][*i + 1*])* a.append*(*ACopy*[*n - 1*][*n - 2*])* b.append*(*ACopy*[*n - 1*][*n - 1*])* c.append*(*0*)* # creating P and Q  
 p = *[*-c*[*0*]* / b*[*0*]]* q = *[*d*[*0*]* / b*[*0*]]* for i in range*(*1, n - 1*)*:  
 p.append*(*- c*[*i*]* / *(*b*[*i*]* + a*[*i*]* \* p*[*i - 1*]))* q.append*((*d*[*i*]* - a*[*i*]* \* q*[*i - 1*])* / *(*b*[*i*]* + a*[*i*]* \* p*[*i - 1*]))* ans = *[[*0 for \_ in range*(*n*)]* for \_ in range*(*n*)]* ans*[*n - 1*]* = *(*d*[*n - 1*]* - a*[*n - 1*]* \* q*[*n - 2*])* / *(*b*[*n - 1*]* + a*[*n - 1*]* \* p*[*n - 2*])* for i in range*(*n - 2, -1, -1*)*:  
 ans*[*i*]* = p*[*i*]* \* ans*[*i + 1*]* + q*[*i*]* return ans  
  
  
c = *[*0*]* + TDMAsolver*(*A, \_d*)*a = f\_i*[*:-1*]*b = *[(*f\_i*[*i*]* - f\_i*[*i - 1*])* / h*[*i*]* - *(*1 / 3*)* \* h*[*i*]* \* *(*c*[*i*]* + 2 \* c*[*i - 1*])* for i in range*(*1, n*)]*b.append*((*f\_i*[*n*]* - f\_i*[*n - 1*])* / h*[*n*]* - *(*2 / 3*)* \* h*[*n*]* \* c*[*n - 1*])*d = *[(*c*[*i*]* - c*[*i - 1*])* / *(*3 \* h*[*i*])* for i in range*(*1, n*)]*d.append*(*-c*[*n - 1*]* / *(*3 \* h*[*n*]))*# print("a:", a)  
# print("b:", b)  
# print("c:", c)  
# print("d:", d)  
  
  
def S*(*x, a, b, c, d, X*)*:  
 for i in range*(*1, len*(*X*))*:  
 if x >= X*[*i - 1*]* and x <= X*[*i*]*:  
 return a*[*i - 1*]* + b*[*i - 1*]* \* *(*x - X*[*i - 1*])* + c*[*i - 1*]* \* *(*x - X*[*i - 1*])* \*\* 2 + d*[*i - 1*]* \* *(*x - X*[*i - 1*])* \*\* 3

**Результат:**

Значение в точке X\*: 1.5862379464285716

|  |
| --- |
|  |
|  |

## Метод наименьших квадратов

Пусть задана таблично в узлах функция . При этом значения функции определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени , у которого неизвестны коэффициенты , . Неизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции.

Минимума можно добиться только за счет изменения коэффициентов многочлена . Необходимые условия экстремума имеют вид

Эту систему для удобства преобразуют к следующему виду:

Эта система называется нормальной системой метода наименьших квадратов (МНК) и представляет собой систему линейных алгебраических уравнений относительно коэффициентов . Решив систему, построим многочлен , приближающий функцию и минимизирующий квадратичное отклонение.

**Входные данные:**

**Изображение выглядит как текст, линия, число, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.**

**Метод наименьших квадратов:**

def mnk\_func*(*x, a*)*:  
 result = 0  
 for i, ai in enumerate*(*a*)*:  
 result += ai \* *(*x \*\* i*)* return result  
  
  
def sum\_sq\_er*(*Xi, Yi, a, f*)*:  
 sum = 0  
 for i, y in enumerate*(*Yi*)*:  
 sum += *(*y - f*(*Xi*[*i*]*, a*))* \*\* 2  
 return sum  
  
  
def get\_a*(*k, Xi, Yi*)*:  
 F = *[]* for j, x in enumerate*(*Xi*)*:  
 ph = *[*x \*\* i for i in range*(*k + 1*)]* F.append*(*ph*)* G = *[[*0*]* \* *(*k + 1*)* for \_ in range*(*k + 1*)]* for i in range*(*k + 1*)*:  
 for j in range*(*k + 1*)*:  
 G*[*i*][*j*]* = sum*(*F*[*m*][*i*]* \* F*[*m*][*j*]* for m in range*(*len*(*Xi*)))* z = *[*sum*(*F*[*m*][*i*]* \* Yi*[*m*]* for m in range*(*len*(*Xi*)))* for i in range*(*k + 1*)]* LU = lu\_decomposition*(*G*)* n = len*(*LU*)* L = *[[*0*]* \* n for \_ in range*(*n*)]* U = *[[*0*]* \* n for \_ in range*(*n*)]* for i in range*(*n*)*:  
 for j in range*(*i + 1*)*:  
 L*[*i*][*j*]* = LU*[*i*][*j*]* for j in range*(*i, n*)*:  
 U*[*i*][*j*]* = LU*[*i*][*j*]* y = forward\_substitution*(*L, z*)* a = backward\_substitution*(*U, y*)* return a

**Результат:**

|  |
| --- |
|  |
|  |

Сумма квадратов ошибок для приближающего многочлена 1-й степени:0.7884327367619048

Сумма квадратов ошибок для приближающего многочлена 2-й степени:0.7747187737857143

## Численное дифференцирование

Формулы численного дифференцирования в основном используются при нахождении производных от функции , заданной таблично. Исходная функция на отрезках, заменяется некоторой приближающей, легко вычисляемой функцией , где – остаточный член приближения, – набор коэффициентов, вообще говоря, различный для каждого из рассматриваемых отрезков, и полагают, что . Наиболее часто в качестве приближающей функции берется интерполяционный многочлен , а производные соответствующих порядков определяются дифференцированием многочлена.

При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой. В этом случае:

При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:

При равностоящих точках разбиения, данная формула обеспечивает второй порядок точности.

Для вычисления второй производной, необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем

**Входные данные: Изображение выглядит как текст, число, линия, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.**

**Дифференциал первого порядка:**

def P1*(*x,X,Y*)*:  
 p\_left, p\_right, ind = approximation\_left\_right*(*x,X,Y*)* return p\_left + *(*p\_right - p\_left*)* \* *(*x - X*[*ind - 1*]* + x - X*[*ind*])* / *(*X*[*ind + 1*]* - X*[*ind - 1*])*

**Дифференциал второго порядка:**

def P2*(*x,X,Y*)*:  
 p\_left, p\_right, ind = approximation\_left\_right*(*x,X,Y*)* return *(*p\_right - p\_left*)* \* 2 / *(*X*[*ind + 1*]* - X*[*ind - 1*])*

**Результат:**

Левосторонняя производная: -5.498666666666667

Правосторонняя производная: -2.0283333333333324

Первая производная в точке X\* : -3.7634999999999996

Вторая производная в точке X\* : 11.56777777777778

## Численное интегрирование

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл не удается. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен.

При использовании интерполяционных многочленов различной степени, получают формулы численного интегрирования различного порядка точности.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку , получим **формулу прямоугольников**:

В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию многочленом Лагранжа первой степени.

Эта формула носит название **формулы трапеций**.

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования: .

Для случая , получим **формулу Симпсона**:

**Метод Рунге‑Ромберга-Ричардсона** позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла порядка точности p на сетке с шагом и на сетке с шагом , то

**Входные данные: **

**Метод прямоугольника:**

def rectangle\_method*(*x0, xk, h*)*:  
 intg = 0  
 xi1 = x0  
 xi2 = x0 + h  
 while *(*xi2 <= xk*)*:  
 intg += task\_function\_5*((*xi1 + xi2*)*/2*)* \* h  
 xi1 += h  
 xi2 += h  
 return intg

**Метод трапеции:**

def trapezoida\_method*(*x0, xk, h*)*:  
 intg = 0  
 xi = x0  
 while *(*xi < xk*)*:  
 intg += *(*task\_function\_5*(*xi*)* + task\_function\_5*(*xi + h*))* \* h / 2  
 xi += h  
 return intg

**Метод Симпсона:**

def simpson*(*a, b, h, f*)*:  
 n = int*((*b - a*)* / h*)* if n % 2 != 0:  
 raise ValueError*(*"Количество интервалов должно быть четным. Попробуйте изменить шаг h."*)* result = f*(*a*)* + f*(*b*)* for i in range*(*1, n*)*:  
 x = a + i \* h  
 if i % 2 == 0:  
 result += 2 \* f*(*x*)* else:  
 result += 4 \* f*(*x*)* return result \* h / 3

**Метод Рунге‑Ромберга-Ричардсона:**

def runge\_romberg\_method*(*x0, xk, h, r, num*)*:  
 if num == 1:  
 p = 1  
 return *(*rectangle\_method*(*x0, xk, h*)* - rectangle\_method*(*x0, xk, h \* r*))* / *(*r \*\* p - 1*)* elif num == 2:  
 p = 2  
 return *(*trapezoida\_method*(*x0, xk, h*)* - trapezoida\_method*(*x0, xk, h \* r*))* / *(*r \*\* p - 1*)* elif num == 3:  
 p = 4  
 return *(*simpson*(*x0, xk, h, task\_function\_5*)* - simpson*(*x0, xk, h \* r,task\_function\_5*))* / *(*r \*\* p - 1*)*

**Результат:**

Метод прямоугольника

c шагом 1.0: -0.1878310847553158

c шагом 0.5: -0.19940628065257357

Метод трапеций

c шагом 1.0: -0.23658183921341816

c шагом 0.5: -0.212206461984367

Метод Симпсона

c шагом 1.0: -0.2071717755928282

c шагом 0.5: -0.20408133624134991

===== Погрешность Метод Рунге-Ромберга =====

Метод прямоугольников:

Для шага 1.0: -0.03947943640366744

Для шага 0.5: -0.011575195897257778

Метод трапеций:

Для шага 1.0: 0.029410063620589927

Для шага 0.5: 0.008125125743017056

Метод Симпсона:

Для шага 1.0: 0.0006246385193753608

Для шага 0.5: 0.00020602929009855275

===== Уточнение значений Метод Рунге-Ромберга =====

Метод прямоугольников:

Для шага 1.0: -0.22731052115898323

Для шага 0.5: -0.21098147654983135

Метод трапеций:

Для шага 1.0: -0.20717177559282823

Для шага 0.5: -0.20408133624134994

Метод Симпсона:

Для шага 1.0: -0.20654713707345285

Для шага 0.5: -0.20387530695125136Лабораторная работа №4

## Численные методы решения задачи Коши

**Введение:**

Рассматривается задача Коши для одного дифференциального уравнения первого порядка разрешенного относительно производной

Требуется найти решение на отрезке , где .

Введем разностную сетку на отрезке : .

Точки называются узлами разностной сетки, расстояния между узлами —*шагом разностной сетки*, а совокупность значений какой-либо величины, заданных в узлах сетки называется *сеточной функцией*.

Приближенное решение задачи Коши будем искать численно в виде сеточной функции.

**Метод Эйлера**

Метод Эйлера играет важную роль в теории численных методов решения ОДУ, хотя и не часто используется в практических расчетах из-за невысокой точности. Вывод расчетных соотношений для этого метода может быть произведен несколькими способами: с помощью геометрической интерпретации, с использованием разложения в ряд Тейлора, конечно, разностным методом (с помощью разностной аппроксимации производной), квадратурным способом (использованием эквивалентного интегрального уравнения).

Рассмотрим вывод соотношений метода Эйлера геометрическим способом. Решение в узле известно из начальных условий. Рассмотрим процедуру получения решения в узле .

График функции , которая является решением задачи Коши, представляет собой гладкую кривую, проходящую через точку , согласно условию , и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точке и равен значению правой части дифференциального уравнения в точке согласно выражению . В случае небольшого шага разностной сетки h график функции и график касательной не успевают сильно разойтись друг от друга и можно в качестве значения решения в узле принять значение касательной , вместо значения неизвестного точного решения . При этом допускается погрешность геометрически представленная отрезком CD. Из прямоугольного треугольника ABC находим СВ=ВА tg(CAB) или . Учитывая, что и заменяя производную на правую часть дифференциального уравнения, получаем соотношение . Считая теперь точку начальной и повторяя все предыдущие рассуждения, получим значение в узле .

Переход к произвольным индексам дает формулу метода Эйлера:

**Методы Рунге-Кутты**

Общая схема методов

Методы Рунге-Кутты представляют собой семейство итерационных методов решения ОДУ. Общая формула для -стадийного метода имеет вид:

где

Коэффициенты , , выбираются так, чтобы обеспечить нужный порядок точности метода.

**Метод Рунге-Кутты 4-го порядка**

Наиболее распространенным является метод 4-го порядка:

где

**Метод Адамса**

При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

Метод Адамса как и все многошаговые методы не является самостартующим, то есть для того, что бы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле решение известно из начальных условий, а в других трех узлах решения можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

**Решение задачи Коши для системы обыкновенных дифференциальных уравнений:**

Рассматривается задача Коши для системы дифференциальных уравнений первого порядка разрешенных относительно производной

К системе дифференциальных уравнений можно применить все методы рассмотренные выше.

Уравнения решаются по порядку.

Для задачи Коши 2-го порядка можно применить следующее разложение в систему (используя замену :

**Входные данные:**

|  |  |
| --- | --- |
| ,  ,  , |  |

**Метод Эйлера:**

def euler\_method*(*f, g, l, r, h, y0, z0*)*:  
 n = int*((*r - l*)* / h*)* + 1  
 x = *[*0*]* \* n  
 y = *[*0*]* \* n  
 z = *[*0*]* \* n  
 x*[*0*]* = l  
 y*[*0*]* = y0  
 z*[*0*]* = z0  
 for i in range*(*n - 1*)*:  
 x*[*i + 1*]* = x*[*i*]* + h  
 y*[*i + 1*]* = y*[*i*]* + h \* f*(*x*[*i*]*, y*[*i*]*, z*[*i*])* z*[*i + 1*]* = z*[*i*]* + h \* g*(*x*[*i*]*, y*[*i*]*, z*[*i*])* return x, y, z

**Метод Рунге-Кутты:**

def runge\_kutta\_method*(*f, g, l, r, h, y0, z0*)*:  
 n = int*((*r - l*)* / h*)* + 1  
 x = *[*0*]* \* n  
 y = *[*0*]* \* n  
 z = *[*0*]* \* n  
 x*[*0*]* = l  
 y*[*0*]* = y0  
 z*[*0*]* = z0  
 for i in range*(*n - 1*)*:  
 K1 = h \* f*(*x*[*i*]*, y*[*i*]*, z*[*i*])* L1 = h \* g*(*x*[*i*]*, y*[*i*]*, z*[*i*])* K2 = h \* f*(*x*[*i*]* + h / 2, y*[*i*]* + K1 / 2, z*[*i*]* + L1 / 2*)* L2 = h \* g*(*x*[*i*]* + h / 2, y*[*i*]* + K1 / 2, z*[*i*]* + L1 / 2*)* K3 = h \* f*(*x*[*i*]* + h / 2, y*[*i*]* + K2 / 2, z*[*i*]* + L2 / 2*)* L3 = h \* g*(*x*[*i*]* + h / 2, y*[*i*]* + K2 / 2, z*[*i*]* + L2 / 2*)* K4 = h \* f*(*x*[*i*]* + h, y*[*i*]* + K3, z*[*i*]* + L3*)* L4 = h \* g*(*x*[*i*]* + h, y*[*i*]* + K3, z*[*i*]* + L3*)* dy = *(*K1 + 2 \* K2 + 2 \* K3 + K4*)* / 6  
 dz = *(*L1 + 2 \* L2 + 2 \* L3 + L4*)* / 6  
 x*[*i + 1*]* = x*[*i*]* + h  
 y*[*i + 1*]* = y*[*i*]* + dy  
 z*[*i + 1*]* = z*[*i*]* + dz  
 return x, y, z

**Метод Адамса:**

def adams\_method*(*f, g, l, r, h, y0, z0*)*:  
 n = int*((*r - l*)* / h*)* + 1  
 x\_start, y\_start, z\_start = runge\_kutta\_method*(*f, g, l, l + 4 \* h, h, y0, z0*)* x = *[*0*]* \* n  
 y = *[*0*]* \* n  
 z = *[*0*]* \* n  
 x*[*:4*]* = x\_start  
 y*[*:4*]* = y\_start  
 z*[*:4*]* = z\_start  
 for i in range*(*3, n - 1*)*:  
 x*[*i + 1*]* = x*[*i*]* + h  
  
 y*[*i + 1*]* = y*[*i*]* + h / 24 \* *(*55 \* f*(*x*[*i*]*, y*[*i*]*, z*[*i*])* -  
 59 \* f*(*x*[*i - 1*]*, y*[*i - 1*]*, z*[*i - 1*])* +  
 37 \* f*(*x*[*i - 2*]*, y*[*i - 2*]*, z*[*i - 2*])* -  
 9 \* f*(*x*[*i - 3*]*, y*[*i - 3*]*, z*[*i - 3*]))* z*[*i + 1*]* = z*[*i*]* + h / 24 \* *(*55 \* g*(*x*[*i*]*, y*[*i*]*, z*[*i*])* -  
 59 \* g*(*x*[*i - 1*]*, y*[*i - 1*]*, z*[*i - 1*])* +  
 37 \* g*(*x*[*i - 2*]*, y*[*i - 2*]*, z*[*i - 2*])* -  
 9 \* g*(*x*[*i - 3*]*, y*[*i - 3*]*, z*[*i - 3*]))* return x, y, z

**Результат:**

|  |
| --- |
|  |
|  |

Погрешность численного решения путем сравнения с точным решением:

Шаг = 0.1

Эйлера: 0.07812580875250488

Рунге-Кутта: 0.0002820390057992516

Адамса: 0.023619295334282944

Шаг = 0.05

Эйлера: 0.034802096090414104

Рунге-Кутта: 1.7495836702423923e-05

Адамса: 0.0024006268887099163

Погрешность методом Рунге – Ромберга:

Эйлера: 0.08183690964830491

Рунге-Кутта: 9.582826047560935e-05

Адамса: 0.008086838571025462

## Численные методы решения краевой задачи для ОДУ

Примером краевой задачи является двухточечная краевая задача для обыкновенного дифференциального уравнения второго порядка.

с граничными условиями, заданными на концах отрезка .

Следует найти такое решение на этом отрезке, которое принимает на концах отрезка значения .

Кроме граничных условий первого рода, используются еще условия на производные от решения на концах - граничные условия второго рода:

или линейная комбинация решений и производных – граничные условия третьего рода:

Возможно на разных концах отрезка использовать условия различных типов.

**Метод стрельбы:**

Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи.

Пусть надо решить краевую задачу краевыми условиями 2-го рода на отрезке . Введем переменные

С граничными условиями:

Поскольку значения функции в точке не заданы, будем рассматривать как параметр. Таким образом, формируется задача Коши:

Обозначим решение этой задачи через , и вычислим значение производной на правом конце:

Следующее значение искомого корня определяется по соотношению

**Конечно-разностный метод:**

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке

С граничными условиями 2-го рода

Введем разностную аппроксимацию производных следующим образом:

Граничные условия второго рода аппроксимируются с помощью односторонних разностей:

В левой границе :

В правой границе :

**Входные данные:**

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

**Преобразование**

Для метода стрельбы

В качестве , а в качестве

Для **метода конечных разностей** нужно преобразовать выражение. Вот результат преобразования для моего выражения:

**Метод стрельбы:**

def shooting\_method*(*mu1, mu2, h*)*:  
 a, b = 0, 1  
 z\_a = 0.75 # y'(0)  
 z\_b = *(*math.exp*(*2*)* \* *(*math.e + 2*))* / *((*math.e + 1*)* \*\* 2*)* # y'(1)  
 eps = 1e-6  
  
 all\_paths = *[]* def dy*(*x, y, z*)*: return z  
  
 def dz*(*x, y, z*)*: return *(*2 \* z + math.exp*(*x*)* \* y*)* / *(*math.exp*(*x*)* + 1*)* def run*(*mu*)*:  
 return runge\_kutta*(*a, mu, z\_a, b, h, dz, dy*)* X1, Y1, Z1 = run*(*mu1*)* X2, Y2, Z2 = run*(*mu2*)* all\_paths.append*((*X1, Y1*))* all\_paths.append*((*X2, Y2*))* phi1 = Z1*[*-1*]* - z\_b  
 phi2 = Z2*[*-1*]* - z\_b  
  
 while abs*(*phi2*)* > eps:  
 dphi = *(*phi2 - phi1*)* / *(*mu2 - mu1*)* mu1, phi1 = mu2, phi2  
 mu2 = mu2 - phi2 / dphi  
 Xk, Yk, Zk = run*(*mu2*)* phi2 = Zk*[*-1*]* - z\_b  
 all\_paths.append*((*Xk, Yk*))* return mu2, Xk, Yk, all\_paths

**Конечно-разностный метод:**

def finite\_difference*(*N*)*:  
 a, b = 0.0, 1.0  
 h = *(*b - a*)* / N  
 x = *[*a + i\*h for i in range*(*N+1*)]* # Функции коэффициентов  
 def A*(*x*)*: return math.exp*(*x*)* + 1  
 def B*(*x*)*: return -2  
 def C*(*x*)*: return -math.exp*(*x*)* def F*(*x*)*: return 0.0  
  
 # Граничные условия  
 alpha = 3/4  
 beta = *(*math.exp*(*2*)* \* *(*math.exp*(*1*)* + 2*))* / *(*math.exp*(*1*)* + 1*)*\*\*2  
  
 # Построение матрицы и правой части  
 size = N + 1  
 A\_matrix = *[[*0.0 for \_ in range*(*size*)]* for \_ in range*(*size*)]* d = *[*0.0 for \_ in range*(*size*)]* # Левая граница: y'(0) ≈ (-3y0 + 4y1 - y2)/(2h) = alpha  
 A\_matrix*[*0*][*0*]* = -3 / *(*2\*h*)* A\_matrix*[*0*][*1*]* = 4 / *(*2\*h*)* A\_matrix*[*0*][*2*]* = -1 / *(*2\*h*)* d*[*0*]* = alpha  
  
 # Внутренние узлы  
 for i in range*(*1, N*)*:  
 xi = x*[*i*]* Ai = A*(*xi*)* Bi = B*(*xi*)* Ci = C*(*xi*)* Fi = F*(*xi*)* A\_matrix*[*i*][*i-1*]* = Ai / h\*\*2 - Bi / *(*2\*h*)* A\_matrix*[*i*][*i*]* = -2 \* Ai / h\*\*2 + Ci  
 A\_matrix*[*i*][*i+1*]* = Ai / h\*\*2 + Bi / *(*2\*h*)* d*[*i*]* = Fi  
  
 # Правая граница: y'(1) ≈ (3yN - 4y\_{N-1} + y\_{N-2})/(2h) = beta  
 A\_matrix*[*N*][*N*]* = 3 / *(*2\*h*)* A\_matrix*[*N*][*N-1*]* = -4 / *(*2\*h*)* A\_matrix*[*N*][*N-2*]* = 1 / *(*2\*h*)* d*[*N*]* = beta  
  
  
 y = solve\_system*(*A\_matrix,d*)* return x, y

**Результат:**

|  |
| --- |
|  |
|  |

Метод стрельбы:

Ошибка Рунге–Ромберга: 0.421791446983601

Ошибка в сравнении с точным решением: 6.050556073768458e-10

Метод конечных разностей:

Ошибка Рунге–Ромберга: 2.5543775514336176e-05

Ошибка в сравнении с точным решением: 1.530734641632098e-05