**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет прикладной математики и физики

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Численные методы»

**Отчет по лабораторным работам по курсу “Численные методы”**.

Студент: Чекушкин Д.И.

Группа: М80-304Б

Преподаватель: Иванов И.Э.

Дата:

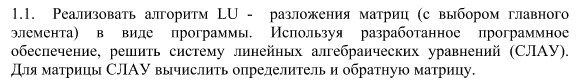
Оценка:

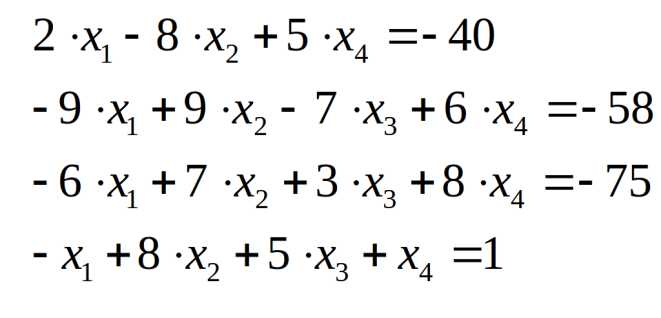
Москва, 2019

Лабораторная работа №1

1.1 Решение СЛАУ методом Гаусса с использованием LU-разложения.

Задание





Теория

LU-разложение - представление матрицы A в виде произведения двух матриц, A=LU, где L — нижняя треугольная матрица, а U — верхняя треугольная матрица.

LU-разложение используется для решения систем линейных уравнений, обращения матриц и вычисления определителя. LU-разложение существует только в том случае, когда матрица A обратима, а все главные миноры матрицы A невырождены.

Этот метод является одной из разновидностей метода Гаусса.

### Решение систем линейных уравнений:

### Полученное LU-разложение матрицы A (матрица коэффициентов системы) может быть использовано для решения семейства систем линейных уравнений с различными векторами b в правой части:

### Ax=b

### Если известно LU-разложение матрицы A, A=LU, исходная система может быть записана как

### LUx=b.

### Эта система может быть решена в два шага. На первом шаге решается система

### Ly=b.

### Поскольку L — нижняя треугольная матрица, эта система решается непосредственно прямой подстановкой.

### На втором шаге решается система

### Ux=y.

### Поскольку U — верхняя треугольная матрица, эта система решается непосредственно обратной подстановкой.

Код программы

import numpy as np

import math

def gauss(A):

print(A)

n = len(A)

for i in range(0, n):

# Находим макс в каждой колонке

maxEl = abs(A[i][i])

maxRow = i

for k in range(i+1, n):

if abs(A[k][i]) > maxEl:

maxEl = abs(A[k][i])

maxRow = k

# Перемещаем вверх

for k in range(i, n+1):

tmp = A[maxRow][k]

A[maxRow][k] = A[i][k]

A[i][k] = tmp

# Обнуляем все элементы под ним

for k in range(i+1, n):

c = -A[k][i]/A[i][i]

for j in range(i, n+1):

if i == j:

A[k][j] = 0

else:

A[k][j] += c \* A[i][j]

# Обратный ход, находим иксы

x = [0 for i in range(n)]

for i in range(n-1, -1, -1): # i = 3 -> 2 -> 1 -> 0

x[i] = A[i][n]/A[i][i]

for k in range(i-1, -1, -1):

A[k][n] -= A[k][i] \* x[i]

return x

#вычисление миноров для определителей высших порядков

def getMinor(m,i,j):

return [row[0:j] + row[j+1:len(m)] for row in (m[0:i]+m[i+1:len(m)])]

def Det(m):

if len(m) == 2:

return m[0][0]\*m[1][1]-m[0][1]\*m[1][0]

det = 0

for c in range(len(m)):

det += ((-1)\*\*c)\*m[0][c]\*Det(getMinor(m,0,c))

return det

#обратная

def InverseMtx(m):

determinant = Det(m)

#случай 2х2

if len(m) == 2:

return [[m[1][1]/determinant, -1\*m[0][1]/determinant],

[-1\*m[1][0]/determinant, m[0][0]/determinant]]

#матрица миноров

minorMtrx = []

for r in range(len(m)):

minorMtrxRow = []

for c in range(len(m)):

minor = getMinor(m,r,c)

minorMtrxRow.append(((-1)\*\*(r+c)) \* Det(minor))

minorMtrx.append(minorMtrxRow)

minorMtrx = np.matrix(minorMtrx).transpose()

minorMtrx /=determinant #делим все элементы на определитель

return minorMtrx

f = open('test1.txt','r')

matrixSize = int(f.readline())

matrix = [list(map(float, f.readline().split())) for i in range(matrixSize)]

vec = list(map(float, f.readline().split()))

print("Матрица СЛАУ:")

print(matrix)

print("Вектор правых частей СЛАУ:")

print(vec)

#соединяем матрицу с правыми частями

C = [row + [element] for row, element in zip(matrix, vec)]

result = np.matrix(gauss(C)).transpose()

print("Ответ:")

print(result)

print("Определитель матрицы равен:")

print(Det(matrix))

print("Обратная матрица:")

print(np.matrix(InverseMtx(matrix)))

f.close()

Результат выполнения

Матрица СЛАУ:

[[2.0, -8.0, 0.0, 5.0], [-9.0, 9.0, -7.0, 6.0], [-6.0, 7.0, 3.0, 8.0], [-1.0, 8.0, 5.0, 1.0]]

Вектор правых частей СЛАУ:

[-40.0, -58.0, -75.0, 1.0]

[[2.0, -8.0, 0.0, 5.0, -40.0], [-9.0, 9.0, -7.0, 6.0, -58.0], [-6.0, 7.0, 3.0, 8.0, -75.0], [-1.0, 8.0, 5.0, 1.0, 1.0]]

Ответ:

[[ 7.]

[ 3.]

[-2.]

[-6.]]

Определитель матрицы равен:

-2007.0

Обратная матрица:

[[ 0.33183857 0.17488789 -0.39910314 0.48430493]

[ 0.09217738 0.10413553 -0.16641754 0.24564026]

[-0.12406577 -0.1509716 0.2077728 -0.13602392]

[ 0.21474838 0.09666168 -0.10662681 0.19930244]]

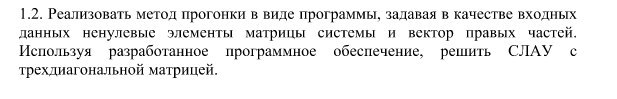
Выводы

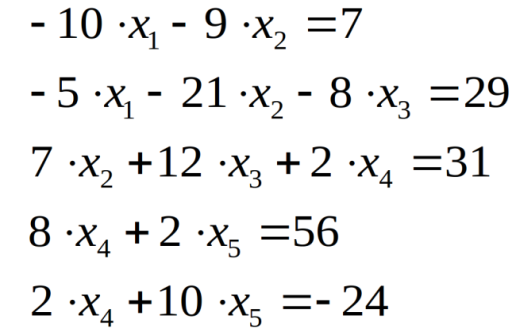
Компьютерная реализация метода Гаусса часто осуществляется с

использованием LU-разложения матриц. В методе Гаусса матрица СЛАУ с помощью равносильных преобразований преобразуется в верхнюю треугольную матрицу, получающуюся в результате прямого хода. В обратном ходе определяются неизвестные. Использовав алгоритм LU-разложения матриц, мы смогли найти неизвестные для заданной матрицы.

### 1.2. Решение СЛАУ методом прогонки

## Задание



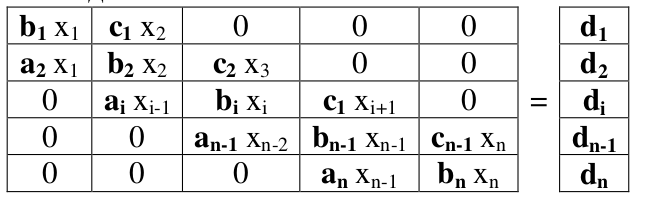


Теория

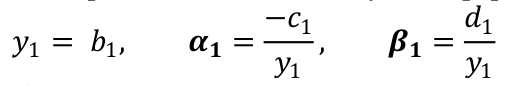
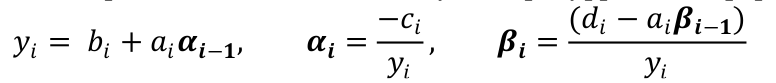
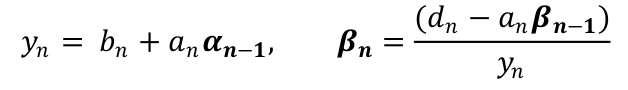
Метод прогонки является частным случаем [метода Гаусса](https://pro-prof.com/forums/topic/gauss-jordan-method) и используется для решения [систем линейных уравнений](https://pro-prof.com/forums/topic/simultaneous-linear-algebraic-equations) вида Ax = B, где A — трёхдиагональная матрица. **Трёхдиагональной матрицей** называется матрица такого вида, где во всех остальных местах, кроме главной диагонали и двух соседних с ней, стоят нули.

Метод прогонки состоит из двух этапов: прямой прогонки и обратной прогонки. На первом этапе определяются прогоночные коэффициенты, а на втором – находят неизвестные x.

## Алгоритм выполнения метода прогонки:

СЛАУ имеет вид:  


Прямая прогонка состоит в вычислении прогоночных коэффициентов αi и βi , где i – номер строки матрицы. Этот этап выполняется при i = 1...n строго по возрастанию значения i.

1. В первой строке матрицы (i = 1) используются формулы:  
   
2. Для строк i от 2 до n-1 используются рекуррентные формулы:  
   
3. При i = n прямая прогонка завершается вычислением:  
   

После этого производится обратная прогонка, в которой происходит вычисление неизвестных xi. Этот этап выполняется при i = n...1 строго по  
убыванию значения i.

1. В последней строке матрицы (i = n) xn = βn.
2. Для всех остальных строк при i от n-1 до 1 применяется формула:  
   xi

Общее число операций в методе прогонки равно 8n+1, т.е. пропорционально числу уравнений. Такие методы решения СЛАУ называют экономичными. Для сравнения число операций в методе Гаусса пропорционально n3.

Для устойчивости метода прогонки достаточно выполнения:

ai ≠ 0 , ci ≠ 0, i=2…n-1

|bi|≥|ai|+|ci|, i=1…n

Код программы

import numpy as np

import math

#Разделяем матрицу на главные линии

def Lines(matrix):

LineA=[]

LineB=[]

LineC=[]

for i in range(1, len(matrix)):

LineA.append(matrix[i][0])

LineC.append(matrix[0][0])#[0,0], потом все [i][1]

for i in range(1, len(matrix)):

LineC.append(matrix[i][1])

LineB.append(matrix[0][1])

for i in range(1, len(matrix) - 1):

LineB.append(matrix[i][2])

return LineA, LineC, LineB

#Вычисляем коэфф-ты

def Coefficients(LineA, LineC, LineB, Ans):

Alpha = [0 for i in range(len(LineC))]

Betta = Alpha.copy()

Alpha[0] = -LineB[0] / LineC[0]

Betta[0] = Ans[0] / LineC[0]

#print('Alpha:',Alpha)

#print('Betta:',Betta)

for i in range(1, len(LineC) - 1):

Alpha[i] = -LineB[i] / (LineC[i] + LineA[i - 1] \* Alpha[i - 1])

Betta[i] = (Ans[i] - LineA[i - 1] \* Betta[i - 1]) / (LineC[i] + LineA[i - 1] \* Alpha[i - 1])

Betta[len(Betta)-1] = (Ans[len(Ans)-1] - LineA[len(LineA)-1] \* Betta[len(Betta)-2]) / (LineC[len(Betta)-1] + LineA[len(LineA)-1] \* Alpha[len(Betta)-2])

return Alpha, Betta

#Обратный ход

def SolveSLAU(Alpha, Betta):

Ans = Betta.copy()

n= len(Betta) - 2

print('N=',n)

while n>=0:

Ans[n] = Alpha[n] \* Ans[n + 1] + Betta[n]

n=n-1

return Ans

def TDMA(matrix, vec):

LineA, LineC, LineB = Lines(matrix)

print("Диагональ А:")

print(np.matrix(LineA).transpose())

print("Диагональ С:")

print(np.matrix(LineC).transpose())

print("Диагональ В:")

print(np.matrix(LineB).transpose())

#print("Ответы:")

#print(np.matrix(vec).transpose())

Alpha, Betta = Coefficients(LineA, LineC, LineB, vec)

#print("Коэфф-ты Альфа:")

#print(np.matrix(Alpha).transpose())

#print("Коэфф-ты Бетта:")

#print(np.matrix(Betta).transpose())

result = SolveSLAU(Alpha, Betta)

return result

f = open('test.txt','r')

matrixSize = int(f.readline())

matrix = [list(map(float, f.readline().split())) for i in range(matrixSize)]

print('Исходная матрица:')

for x in matrix: print(x)

vec = list(map(float, f.readline().split()))

#print(vec)

res = TDMA(matrix, vec)

print("Ответ:")

print(np.matrix(res).transpose())

f.close()

Результат выполнения

Исходная матрица:

[-10.0, -9.0]

[-5.0, -21.0, -8.0]

[7.0, 12.0, 2.0]

[0.0, 8.0, 2.0]

[2.0, 10.0]

Диагональ А:

[[-5.]

[ 7.]

[ 0.]

[ 2.]]

Диагональ С:

[[-10.]

[-21.]

[ 12.]

[ 8.]

[ 10.]]

Диагональ В:

[[-9.]

[-8.]

[ 2.]

[ 2.]]

N= 3

Ответ:

[[ 2.]

[-3.]

[ 3.]

[ 8.]

[-4.]]

Выводы

Метод прогонки является одним из эффективных методов решения СЛАУ с трех - диагональными матрицами, возникающих при конечно-разностной аппроксимации задач для обыкновенных дифференциальных уравнений (ОДУ) и уравнений в частных производных второго порядка и является частным случаем метода Гаусса. Использовав метод прогонки, мы решили СЛАУ с трехдиагональной матрицей.

### 1.3. Решение СЛАУ методом простых итераций и методом Зейделя

## Задание

## 

## 

## Теория

## Метод простых итераций:

**Метод простых итераций**

Пусть дана линейная система (1).

|  |  |
| --- | --- |
| http://ok-t.ru/life-prog/baza2/588341428588.files/image002.gif | (1) |

Систему (1) коротко можно записать в виде матричного уравнения:

|  |  |
| --- | --- |
| *Ах*= *b*, | (2) |

Предполагая, что диагональные коэффициенты

*aij* = 0 (*i* = 1, 2, …, *n*),

разрешим первое уравнение системы (1) относительно *х*1, второе – относительно *х*2и т. д. Тогда получим эквивалентную систему

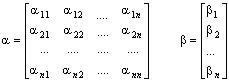
|  |  |
| --- | --- |
| http://ok-t.ru/life-prog/baza2/588341428588.files/image004.gif | (3) |

где

http://ok-t.ru/life-prog/baza2/588341428588.files/image006.gif при *i* ¹ *j*

и a*ij* = 0 при *i* = *j* (*i, j* = 1, 2, …, *n*).

Введя матрицы



Систему (3) можно записать в матричной форме

**x** = b + a***x,***

а любое (*k* + 1) приближение вычисляется по формуле

|  |  |
| --- | --- |
| x (k+1) = b + a*x*(k). | (4) |

Напишем формулы приближений в развернутом виде:

|  |  |
| --- | --- |
| http://ok-t.ru/life-prog/baza2/588341428588.files/image010.gif | (4¢) |

Приведем *достаточное условие сходимости метода итераций.*

**Теорема:***Процесс итерации для приведенной линейной системы*(18)*сходится к единственному ее решению, если какая-нибудь каноническая норма матрицы*a *меньше единицы, т.е. для итерационного процесса*(19)*достаточное условие есть*

|  |  |
| --- | --- |
| http://ok-t.ru/life-prog/baza2/588341428588.files/image012.gif | (5) |

**Следствие 1**. Процесс итерации для системы (3) сходится, если:

1) http://ok-t.ru/life-prog/baza2/588341428588.files/image014.gif < 1 (*m-*норма или неопределенная норма)

или

2) http://ok-t.ru/life-prog/baza2/588341428588.files/image016.gif < 1 (*l-*норма или норма *L*1)

или

3) http://ok-t.ru/life-prog/baza2/588341428588.files/image018.gif < 1 (*k-*норма или Евклидова норма).

**Следствие 2**. Для системы (1) процесс итерации сходится, если выполнены неравенства:

|  |  |
| --- | --- |
| 1) http://ok-t.ru/life-prog/baza2/588341428588.files/image020.gif  или 2) http://ok-t.ru/life-prog/baza2/588341428588.files/image022.gif , |  |

где штрих у знака суммы означает, что при суммировании пропускаются значения *i* = *j*, т. е. сходимость имеет место, если модули диагональных элементов матрицы *А*системы (1) или для каждой строки превышают сумму модулей недиагональных элементов этой строки, или же для каждого столбца превышают сумму модулей недиагональных элементов этого столбца.

Метод Зейделя

*Метод Зейделя* представляет собой некоторую модификацию метода итераций. Основная его идея заключается в том, что при вычислении (*k* + 1)-го приближения неизвестной *xi* учитываются уже вычисленные ранее (*k* + 1)-е приближения неизвестных *x*1, *x*2, …, *xi*- 1.

Пусть получена эквивалентная система (1). Выберем произвольно начальные приближения корней http://old.exponenta.ru/educat/systemat/hanova/equation/images/Image1159.gif. Далее, предполагая, что *k*-ые приближения http://old.exponenta.ru/educat/systemat/hanova/equation/images/Image1160.gif корней известны, согласно Зейделю будем строить (*k* + 1)-е приближения корней по формулам:

|  |  |
| --- | --- |
| http://old.exponenta.ru/educat/systemat/hanova/equation/images/Image1161.gif | (22) |

Заметим, что указанные выше условия сходимости для простой итерации остается верной для итерации по методу Зейделя. Обычно метод Зейделя дает лучшую сходимость, чем метод простой итерации, но приводит к более громоздким вычислениям.

## Код программы

## import numpy as np

## import math

## def Jacobi\_B(matrix, vec):

## B = np.zeros(shape=(len(matrix),len(matrix)))

## for i in range(len(matrix)):

## for j in range(len(matrix)):

## B[i][j]=-matrix[i][j] / matrix[i][i]

## D = np.zeros(len(vec))

## for i in range(len(matrix)):

## D[i] = vec[i] / matrix[i][i]

## for i in range(len(matrix)):

## B[i][i] = 0

## return B, D

## def Jacobi\_Zeidel(matrix, vec, epsilon, isJacobi):

## martixB, vecD = Jacobi\_B(matrix, vec)

## vecD1 = np.matrix(vecD.copy()).transpose()

## matrixB1 = np.matrix(martixB)

## x\_n = vecD1.copy()

## x\_nn = x\_n.copy()

## matrixNorm = np.linalg.norm(matrix)

## iterCounter = 0

## if isJacobi:

## coeff = matrixNorm / (1 - matrixNorm)

## else:

## coeff = np.linalg.norm(np.triu(matrixB1), np.inf) / (1 - np.linalg.norm(matrixB1, np.inf))

## while True:

## if isJacobi:

## #Расчетная формула метода простой итерации:

## #x(n+1)=Bx(x)+d

## x\_nn = np.dot(matrixB1, x\_n)+vecD1

## else:

## # Зейдель

## #x\_i(n+1)=b\_{i1}x\_{1}(n+1)+b\_{i2}x\_{2}(n+1)+...+b\_{im}x\_{m}(n)+d\_{i}

## #i=1, 2,...m..

## for i in range(len(vecD1)):

## x\_nn[i][0] = vecD1.item(i, 0) + math.fsum([matrixB1.item(i, j) \* x\_n.item(j, 0) for j in range(len(vecD1))])

## #проверяем условие окончания

## if (Endfunc(x\_n, x\_nn, coeff)<= epsilon):

## break

## x\_n = x\_nn.copy()

## iterCounter += 1

## return x\_nn, iterCounter

## def Endfunc(firstVec, secondVec, coeff):

## k = np.linalg.norm(secondVec - firstVec, np.inf)

## return k \* coeff

## f = open('test.txt','r')

## epsilonision = float(f.readline())

## matrixSize = int(f.readline())

## matrix = [list(map(float, f.readline().split())) for i in range(matrixSize)]

## vec = list(map(float, f.readline().split()))

## print("Метод простых итераций:")

## Jacobi\_result, Jacobi\_iter = Jacobi\_Zeidel(matrix, vec, epsilonision, isJacobi=True)

## print('Ответ', Jacobi\_result)

## print('Количество итераций', Jacobi\_iter+1)

## print("Метод Зейделя:")

## Zeidel\_result, Zeidel\_iter = Jacobi\_Zeidel(matrix, vec, epsilonision, isJacobi=False)

## print('Ответ', Zeidel\_result)

## print('Количество итераций', Zeidel\_iter+1)

## Результат выполнения

## Метод простых итераций:

## Ответ [-6.00000813e+00 -1.00000018e+00 -0.99955967e-01 -0.00034007e-05]

## Количество итераций = 14

## Метод Зейделя:

## Ответ [-5.99985423e+00 -1.00008159e+00 -0.99922539e-01 0.00036166e-06]

## Количество итераций = 5

Выводы

Метод простых итераций

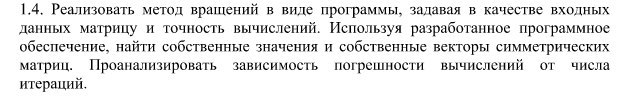
Очевидно преимущество итерационных методов по сравнению, например, с рассмотренным выше методом Гаусса. В вычислительном процессе участвуют только произведения матрицы на вектор, что позволяет работать только с ненулевыми элементами матрицы, значительно упрощая процесс хранения и обработки матриц.

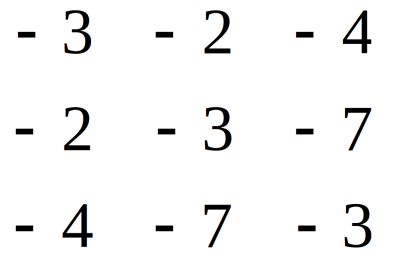
Метод Зейделя

Таким образом, метод Зейделя является методом простых итераций с матрицейправых частей α = (E − B)−1C и вектором правых частей (E − B)−1β и, следовательно,сходимость и погрешность метода Зейделя можно исследовать с помощью формул,выведенных для метода простых итераций, в которых вместо матрицы α подставленаматрица(E − B)−1C, а вместо вектора правых частей – вектор(E − B)−1β.

### 1.4. Решение СЛАУ методом вращений

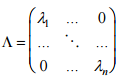
## Задание





Теория:

Метод вращений основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия Λ = U−1AU, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной (U-1 = UT), то Λ = UTAU, где Λ - диагональная матрица с собственными значениями на главной диагонали



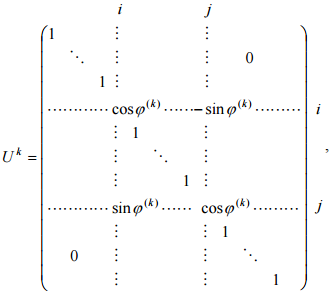
Пусть дана симметрическая матрица A. Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения следующий:

Пусть известна матрица A(k) на k–й итерации, при этом для k=0 A(0) = A.

1. 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:



В матрице вращения на пересечении i −й строки и j − го столбца находится элемент где ϕ(k)- угол вращения, подлежащий определению. Симметрично относительно главной диагонали (j-я строка, i-й столбец) расположен элемент ; Диагональные элементыиравны соответственнодругие диагональные элементы

остальные элементы в матрице вращения U(k) равны нулю.

Угол вращения ϕ(k)определяется из условия:



причем если 

1. Строится матрица A(k+1)



в которой элемент

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



Если t(A(k+1))> ε то итерационный процесc



продолжается. Если t(A(k+1)) < ε, то итерационный процесс останавливается, и в качестве искомых собственных значений принимаются.

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы U = U(0)U(1)…U(k), т.е.



причем эти собственные векторы будут ортогональны между собой, т.е.

Код программы

import math

import numpy as np

def new\_matrix(matrixSize):

count = 0

matrix = []

for i in range(matrixSize):

matrix.append([])

for j in range(matrixSize):

matrix[i].append([])

return matrix

#транспонировать матрицу

def transpose(matrix, matrixSize):

temp = 0

for i in range(matrixSize):

for j in range(i):

temp = matrix[i][j]

matrix[i][j] = matrix[j][i]

matrix[j][i] = temp

return matrix

#умножить матрицу

def multiply(first, second, matrixSize):

temp = 0

third = new\_matrix(matrixSize)

for i in range(matrixSize):

for j in range(matrixSize):

temp = 0

for l in range(matrixSize):

temp = temp + first[i][l]\*second[l][j]

third[i][j] = temp

return third

#создать еденичную

def imatrix(matrixSize):

matrix = new\_matrix(matrixSize)

for i in range(matrixSize):

for j in range(matrixSize):

if(i == j):

matrix[i][j] = 1;

else:

matrix[i][j] = 0;

return matrix

#найти максимальный элемент

def search\_max(matrix, matrixSize):

max = 0

max\_i = 0

max\_j = 0

for i in range(matrixSize):

for j in range(matrixSize):

if(i == j):

continue

if(abs(matrix[i][j]) > max):

max = matrix[i][j]

max\_i = i

max\_j = j

return [max, max\_i, max\_j]

#Сам метод

def rotating\_method(matrix, matrixSize, epsilon):

max = 0

max\_i = 0

max\_j = 0

fi = 0

cosfi = 0

sinfi = 0

AMatrix = matrix

VecMatrix = matrix

resVec = matrix

i=0

while(1):

maxtemp = search\_max(AMatrix, matrixSize)

max = maxtemp[0]

max\_i = maxtemp[1]

max\_j = maxtemp[2]

#условие выхода

if(abs(max)<=epsilon):

break

# находим угол поворота

fi = math.atan(2.0\*max/(AMatrix[max\_i][max\_i] - AMatrix[max\_j][max\_j]))/2.0

cosfi = math.cos(fi)

sinfi = math.sin(fi)

# формируем матрицу вращения

Hmatrix = imatrix(matrixSize)

Hmatrix[max\_i][max\_i] = cosfi

Hmatrix[max\_j][max\_j] = cosfi

Hmatrix[max\_i][max\_j] = sinfi

Hmatrix[max\_j][max\_i] = -sinfi

#для нахождения собственных векторов перемножаем все матрицы поворота

VecMatrix=Hmatrix

if i<1:

resVec=VecMatrix

print('H\_0',transpose(resVec,matrixSize))

else:

resVec=multiply(resVec,transpose(VecMatrix,matrixSize),matrixSize)

HTmatrix = Hmatrix

HTmatrix = transpose(HTmatrix, matrixSize)

TempMatrix = multiply(Hmatrix, AMatrix, matrixSize)

AMatrix = multiply(TempMatrix, HTmatrix, matrixSize)

i=i+1

print('iterations',i)

matrix = AMatrix

return matrix,resVec

epsilon = float(input())

matrixSize = int(input())

matrix = [list(map(float, input().split())) for i in range(matrixSize)]

resVec = matrix

print(matrix)

print("Метод вращений:")

matrix, resVec = rotating\_method(matrix, matrixSize, epsilon)

print(matrix)

print('Собственные значения:')

for i in range(len( matrix )):

print('matrix[',i,'][',i,']=', matrix[i][i])

print('Собственные векторы')

for i in range(len( matrix )):

print(i+1,'-ый:')

for j in range(len( matrix )):

print(resVec[j][i])

Результат выполнения

[[-3.0, -2.0, -4.0], [-2.0, -3.0, -7.0], [-4.0, -7.0, -3.0]]

Собственные значения:

matrix[ 0 ][ 0 ]= -1.3064112486751172

matrix[ 1 ][ 1 ]= 4.329318261256128

matrix[ 2 ][ 2 ]= -12.022907012581012

Собственные векторы

1 -ый:

[ 0.87397404

0.22628749

0.43007366]

2 -ый:

[-0.46629413

0.63976096

0.61096292]

3 -ый:

[-0.13689108

-0.73450656

0.664651 ]]

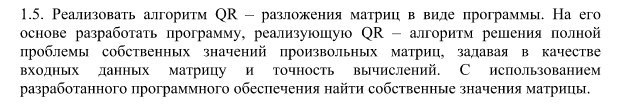
Выводы

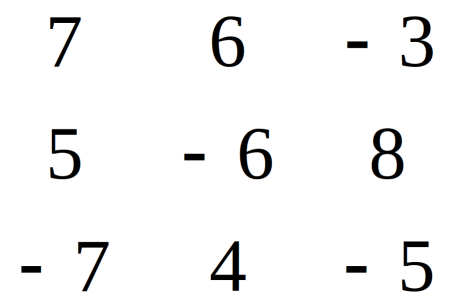
Метод вращений Якоби применим только для симметрических матриц 

(A = AT) и решает полную проблему собственных значений и собственных векторов таких матриц.

### 1.5. QR – алгоритм разложения матриц

## Задание





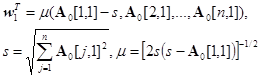
Теория:

 Основная идея этого метода состоит в разложении исходной матрицы **А**0*=***А** в произведение ортогональной матрицы и верхней треугольной. Последовательность преобразований дает нам очередные модификации матрицы **А**: **A***k=***Q***k***R***k*, **A***k*+1*=***R***k***Q***k*, *k=*0, 1, 2, …, где каждая из матриц **Q***k* является ортогональной (если матрица **А** вещественна) или унитарной (если А невещественна). **R***k* – верхние треугольные матрицы.

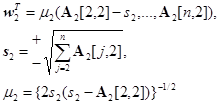
Примечание: *унитарной* называется матрица **U**, удовлетворяющая соотношению **U**\***U***=***E**, где **U**\*– матрица, сопряженная к **U**, т.е. получающаяся из **U** заменой элементов на комплексно-сопряженные с последующим транспонированием, **Е** – единичная матрица.

Для преобразований будем использовать известный *алгоритм Хаусхолдера*, состоящий в следующем:

пусть некоторый вектор ***w***1 выбран так, что



Тогда **A**2=(**E**-2***w***1 http://ok-t.ru/life-prog/baza2/1010836676225.files/image141.png )**A**0=  , где \* обозначает в общем случае ненулевой элемент. Выберем теперь вектор ***w***2 так, что



Лежащие ниже главной диагонали элементы двух первых столбцов матрицы (**E**-2***w***2 http://ok-t.ru/life-prog/baza2/1010836676225.files/image147.png )**A**2 обратятся в нуль. Продолжая этот процесс с помощью векторов ***w****i* c нулями в первых *i*–1 позициях, получим

(**E**-2***w****n*-1 http://ok-t.ru/life-prog/baza2/1010836676225.files/image149.png )…(**E**-2***w***2 http://ok-t.ru/life-prog/baza2/1010836676225.files/image147.png )(**E**-2***w***1 http://ok-t.ru/life-prog/baza2/1010836676225.files/image141.png )**A**=**R**=**QA**

где **R** – верхняя треугольная матрица, а **Q** – ортогональная в силу того, что составляющие ее сомножители в круглых скобках есть ортогональные матрицы. Так как в этом случае **Q**-1*=***Q**T, то можем записать **A***=***QR**, представляющее собой QR-разложение матрицы **А**. ***Теорема о QR-алгоритме*** звучит так:

Если все собственные значения матрицы различны по абсолютной величине и **A***=***PDP**-1, где **D** – диагональная матрица из собственных значений матрицы **А**, то генерируемые QR-алгоритмом матрицы **A***k* сходятся к верхней треугольной матрице с собственными значениями в главной диагонали, а элементы ниже диагонали сходятся к нулю с линейной скоростью, пропорциональной отношению собственных значений.

Для повышения эффективности приведенного алгоритма перед его применением матрицу рекомендуют привести к так называемой *форме Хессенберга* с нулями ниже первой за главной поддиагональю; это приведение осуществляют с применением преобразований Хаусхолдера. После этого разложение матрицы выполняется значительно проще или уже рассмотренными преобразованиями Хаусхолдера, либо (что еще лучше) воспользоваться *преобразованиями Гивенса* (плоскими вращениями). Но мы эти усовершенствования алгоритма рассматривать не будем. Скажем только, что QR-алгоритм практически не налагает на исходную матрицу существенных ограничений и является численно устойчивым.

Код программы

import numpy as np

import math

#Норма матрицы

def norm(x):

return math.sqrt(sum([x\_i\*\*2 for x\_i in x]))

#Преобразование Хаусхолдера

#компоненты вектора v\_n определим,

#используя элементы n столбца матрицы An

#первый элемент для первого вектора v^1:

#v^1\_1 = a^0\_11+sign(a^0\_11)(norm(a^0\_j1))^(1/2) j = 1..

def householder(A, i):

v = np.zeros(len(A))

v[i] = A[i] + np.sign(A[i]) \* norm(A[i:])

for i in range(i + 1, len(A)):

v[i] = A[i]

h = np.eye(len(A)) - (2 / np.dot(v , v)) \* np.dot(v[:, None] , v[None, :])

return h

#QR- разложение

#Q = H1 x H2 x ...

#R = A\_{n-1} = H\_{n-1} x A\_{n-2}

def qr(matrix):

matrixQ = np.eye(len(matrix))

matrixR = matrix.copy()

for i in range(len(matrix) - 1):

matrixH = householder(matrixR[:, i], i)

matrixQ = np.dot(matrixQ , matrixH)

matrixR = np.dot(matrixH , matrixR)

return matrixQ, matrixR

#a[1][1] и пара комплесно-сопряженных значений, которая находится из квадратного уравнения

def eigens(mtx, idx):

matrix = mtx.tolist()

a11 = matrix[idx][idx]

a12 = matrix[idx][idx + 1]

a21 = matrix[idx + 1][idx]

a22 = matrix[idx + 1][idx + 1]

return np.roots((1, -a11 - a22, a11 \* a22 - a12 \* a21))

def eigenVal(matrix, epsilon):

A = matrix

res = []

counter = 0

check = 0

eps = epsilon

while True:

matrixQ, matrixR = qr(A)

result = []

idx = 0

print("итерация №:", counter)

print("матрица Q:", matrixQ)

print("матрица R", matrixR)

A = np.dot(matrixR , matrixQ)

### Условие окончания

while idx < len(A):

if norm(A[idx + 1:, idx]) <= eps:

result.append(A.item(idx, idx))

elif norm(A[idx + 2:, idx]) <= eps:

roots = eigens(A, idx)

result.extend(roots)

idx += 1

idx += 1

res = result

###

if len(res) == len(matrix):

if check == 0:

check = 1

else:

break

counter = counter + 1

return res, counter

f = open('test.txt','r')

epsilon = float(f.readline())

matrixSize = int(f.readline())

matrix = np.matrix([list(map(float, f.readline().split())) for i in range(matrixSize)])

vec, it = eigenVal(matrix, epsilon) #it - кол-во итераций

print('Собственные значения:')

for i in range(len( vec )):

print(vec[i])

f.close()

Результат выполнения

итерация №: 0

матрица Q: [[-0.63116874 0.74553831 0.21400616]

[-0.45083482 -0.57713437 0.6809287 ]

[ 0.63116874 0.33329948 0.7003838 ]]

матрица R [[-1.10905365e+01 1.44267142e+00 -4.86901603e+00]

[ 1.63618473e-16 9.26923401e+00 -8.52018728e+00]

[ 2.95371035e-16 6.16802988e-16 1.30349208e+00]]

итерация №: 1

матрица Q: [[-0.32324402 -0.94610578 -0.0199289 ]

[ 0.94282823 -0.32378768 0.07897129]

[-0.08116792 0.00673747 0.99667767]]

матрица R [[-1.01360662e+01 -4.29000187e+00 1.80248174e+00]

[-7.43486085e-17 1.28004642e+01 4.43718149e+00]

[ 1.28543629e-18 1.39227017e-16 1.03278434e+00]]

итерация №: 2

матрица Q: [[-0.07787643 -0.99696079 0.00210891]

[ 0.99693747 -0.07785911 0.00732421]

[-0.00713775 0.00267284 0.99997095]]

матрица R [[ 1.17444499e+01 -4.95812405e+00 5.28006984e+00]

[ 6.36781182e-16 -1.06371924e+01 -2.07494411e+00]

[-9.77562013e-18 6.65776877e-18 1.07261806e+00]]

итерация №: 3

матрица Q: [[-4.86400373e-01 8.73736024e-01 1.93104780e-04]

[-8.73735817e-01 -4.86400074e-01 -8.30466429e-04]

[-6.31682256e-04 -5.72661744e-04 9.99999637e-01]]

матрица R [[ 1.21201455e+01 4.78172507e+00 -6.82242529e-01]

[ 3.43481691e-16 -1.02808801e+01 5.64966920e+00]

[-6.74991561e-19 -9.35371994e-19 1.07539168e+00]]

Собственные значения:

-13.982558578866982

8.907167287684565

1.0753912911824177

Выводы

Суть QR-алгоритма заключается в итерационном приведении матрицы  A к некоторой унитарно подобной ей матрице  AN при помощи QR-разложения. Матрица  AN является правой верхней треугольной (или блочно треугольной) матрицей, а значит ее диагональ содержит собственные значения (в блочном случае собственные значения матрицы являются собственными значениями диагональных блоков). В силу подобия матриц  A и  AN их наборы собственных значений совпадают. Таким образом, задача поиска собственных значений матрицы  A сводится к задаче выведения матрицы  AN и поиска собственных значений для нее.

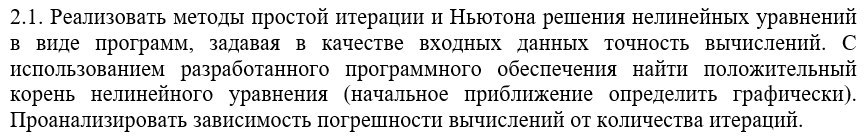
В описываемом алгоритме исходная матрица *хессенбергова*. Благодаря инвариантности хессенберговой формы к QR-алгоритму видно, что так как  A0=A имеет эту форму, то имеют её и все матрицы Ak

.

Лабораторная работа №2

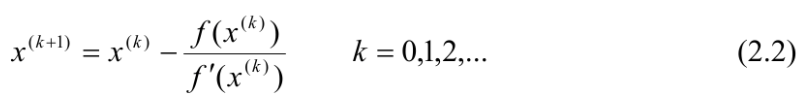
### 2.1 Решение нелинейных уравнений

## Задание



Теория

Метод Ньютона (метод касательных). При нахождении корня уравнения (2.1) методом Ньютона, итерационный процесс определяется формулой



Для начала вычислений требуется задание начального приближения x(0).

Условия сходимости метода определяются следующей теоремой [2]:

**Теорема 2.2.** Пусть на отрезке [a,b] функция f(x) имеет первую и вторую производные постоянного знака и пусть f(a) f(b) < 0 .

Тогда если точка x(0) выбрана на [a,b] так, что

f(x(0) ) f ′′(x(0) ) > 0 , (2.3)

то начатая с нее последовательность x(k)(k = 0,1,2,...) , определяемая методом Ньютона (2.2), монотонно сходится к корню Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.37.30.pngуравнения (2.1).

В качестве условия окончания итераций в практических вычислениях часто используется правило x(k+1) − x(k)< ε Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.36.18.png

Метод простой итерации. При использовании метода простой итерации уравнение(2.1) заменяется эквивалентным уравнением с выделенным линейным членом

x = ϕ(x) (2.5)

Решение ищется путем построения последовательности

x(k+1) = ϕ(x(k )) k = 0,1,2,... (2.6)

начиная с некоторого заданного значения x(0) . Если ϕ(x) - непрерывная функция, а x(k )(k = 0,1,2,...) - сходящаяся последовательность, то значение Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.48.36.pngявляетсярешением уравнения (2.5).

Условия сходимости метода и оценка его погрешности определяются теоремой [2]:

**Теорема 2.3.** Пусть функция ϕ(x) определена и дифференцируема на отрезке [a,b]. Тогда есливыполняются условия:

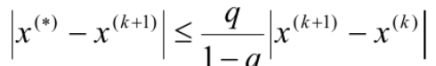
1)Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.53.32.png

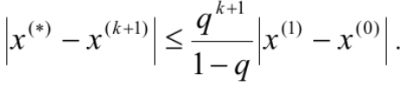
2) Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.53.43.png

то уравнение (2.5) имеет и притом единственный на [a,b] корень x(\*) ;

к этому корню сходится определяемая методом простой итерациипоследовательность x(k)(k = 0,1,2,...) , начинающаяся с любого Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.58.19.png

При этом справедливы оценки погрешности Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 18.00.00.png





Код программы

import matplotlib.pyplot as plt

import numpy as np

import math as m

from math import e

def function(x):

return e\*\*x-x\*\*3+3\*x\*\*2-2\*x-3 # Функция

def first\_derivative(x):

return -3\*x\*\*2+6\*x+e\*\*x-2 # Первая производная

def second\_derivative(x):

return -6\*x+e\*\*x+6 # Вторая производная

def function2(x):

return m.log(x\*\*3.0-3.0\*x\*\*2.0+2.0\*x+3.0)

def function2\_derivative(x):

return (3\*x\*\*2-6\*x+2)/(x\*\*3+2\*x+3) # Ее производная

def newton(function, start\_x, first\_derivative, second\_derivative, eps):

#счетчик итераций

iters = 0

x\_prev = start\_x

x\_cur = x\_prev - function(x\_prev) / first\_derivative(x\_prev)

print("i\t start\_x\t function(start\_x)")

print("x%d\t%e\t%e"%(iters,x\_prev,function(x\_prev)))

iters = 1

while abs(x\_cur - x\_prev) >= eps:

x\_prev = x\_cur

x\_cur = x\_prev - function(x\_prev) / first\_derivative(x\_prev)

print("x%d\t%e\t%e"%(iters,x\_cur,function(x\_cur)))

if abs(x\_prev - x\_cur)<= eps:

plt.plot(x\_prev, function(x\_prev), 'or')

return x\_cur

iters += 1

return x\_cur, iters

def simple\_iter(eps):

begin = 0.9

end = 1.1

q = max([abs(function2\_derivative(x)) for x in np.arange(begin, end, eps)])

coeff = q / (1 - q)

iters = 0

x\_prev = (begin + end) / 2.0

x\_cur = function2(x\_prev)

print("x%d\t%e\t%e"%(iters,x\_prev,function(x\_prev)))

while abs(x\_cur - x\_prev) \* coeff >= eps:

x\_prev = x\_cur

x\_cur = function2(x\_prev)

iters += 1

print("x%d\t%e\t%e"%(iters,x\_cur,function2(x\_cur)))

return x\_cur, iters

solve = newton(function, 1.0, first\_derivative, second\_derivative, 0.001)

print('Методом Ньютона, ответ: '+str(solve))

result, iter\_count = simple\_iter(0.001)

print('Метод итераций, ответ: '+ str(result))

# График

u = np.arange(0, 2, 0.001) # значения Х

f = e\*\*u-u\*\*3+3\*u\*\*2-2\*u-3

plt.plot(u,f)

plt.title('Graphic')

plt.xlabel('X')

plt.ylabel('Y')

plt.grid(True)

plt.legend(['Solve'], loc='upper left')

plt.show()

Результат выполнения

i start\_x function(start\_x)

x0 1.000000e+00 -2.817182e-01

x1 1.073833e+00 4.632837e-06

x2 1.073831e+00 1.741718e-12

Методом Ньютона, ответ: 1.0738314383248424

x0 1.000000e+00 -2.817182e-01

x1 1.065520e+00 1.076626e+00

x2 1.076626e+00 1.072892e+00

x3 1.072892e+00 1.074147e+00

Метод итераций, ответ: 1.0728922365493925

Выводы

Метод Ньютона.

Тогда если точка x(0) выбрана на [a,b] так, что f(x(0) ) f ′′(x(0) ) > 0, (2.3)то начатая с нее последовательность x(k)(k = 0,1,2,...) , определяемая методом Ньютона (2.2), монотонно сходится к корню Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 17.37.30.pngуравнения (2.1).

Метод простой итерации.

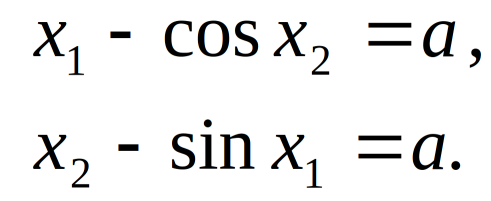
При использовании метода простой итерации уравнение(2.1) заменяется эквивалентным уравнением с выделенным линейным членомx = ϕ(x)(2.5).

### 2.2 Решение систем нелинейных уравнений

## Задание

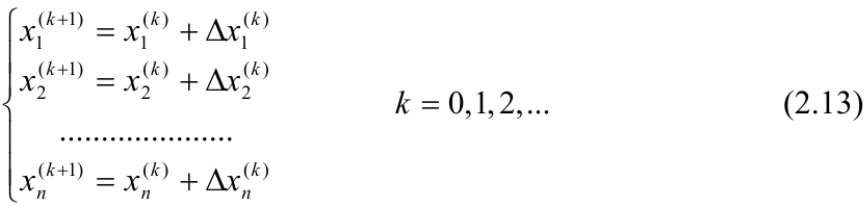
## 

|  |  |
| --- | --- |
| 10 | 1 |
| 11 | 2 |
| 12 | 3 |

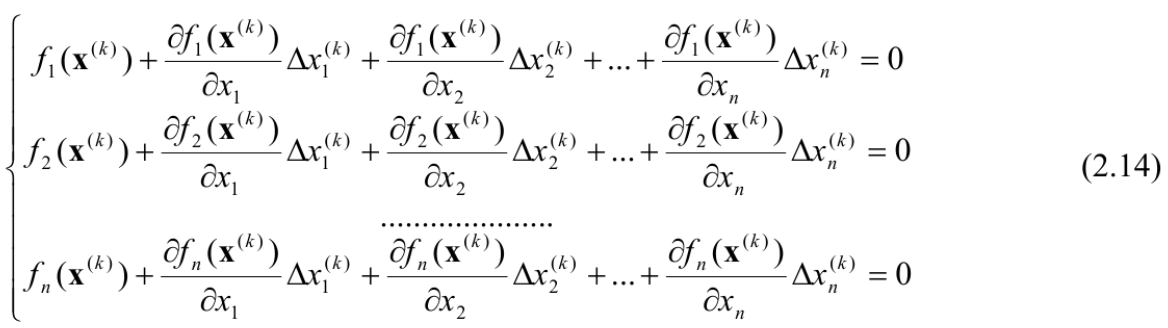


Теория

Метод Ньютона. Если определено начальное приближение Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 19.59.09.png итерационный процесс нахождения решения системы (2.11) методом Ньютона можно представить в виде

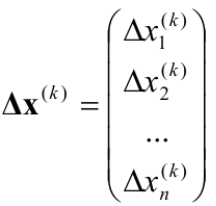


где значения приращений Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 20.01.14.png определяются из решения системы линейных алгебраических уравнений, все коэффициенты которой выражаются через известное предыдущее приближение Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 20.01.52.png



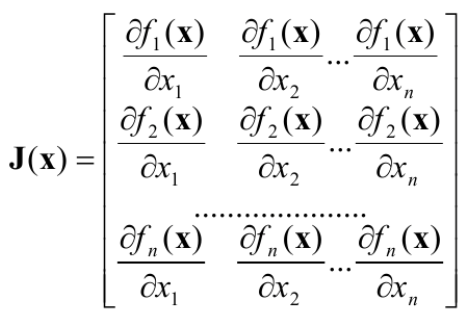
В векторно-матричной форме расчетные формулы имеют вид

x(k+1) = x(k) + Δx(k) k = 0,1, 2,... (2.15)

где вектор приращений находится из решения уравнения

f (x(k) ) + J(x(k) )Δx(k) = 0 (2.16)

Здесь

- матрица Якоби первых производных вектор-

функции f(x).

Выражая из (2.16) вектор приращений Δx(k) и подставляя его в (2.15),итерационный процесс нахождения решения можно записать в виде

x(k+1) = x(k) − J−1 (x(k ))f (x(k)) k = 0,1, 2,... (2.17)

где J−1(x) - матрица, обратная матрице Якоби. Формула (2.17) есть обобщение формулы(2.2) на случай систем нелинейных уравнений.

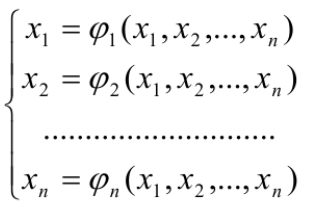
При реализации алгоритма метода Ньютона в большинстве случаев предпочтительным является не вычисление обратной матрицы J−1 (x(k)), а нахождение из системы (2.14) значений приращений Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 20.22.39.pngи вычисление нового приближения по (2.13). Для решения таких линейных систем можно привлекать самые разные методы, как прямые, так и итерационные (см. раздел 1.1), с учетом размерности n решаемой задачи и специфики матриц Якоби J(x) (например, симметрии, разреженности и т.п.).

В практических вычислениях в качестве условия окончания итераций обычноиспользуется критерий [2,5]

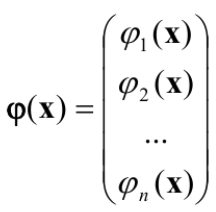
Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 20.22.48.png (2.18)

где ε - заданная точность.

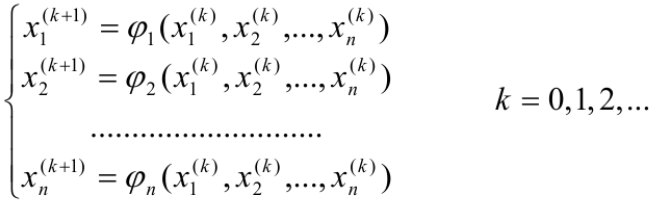
Метод простой итерации.При использовании метода простой итерации системауравнений (2.11) приводится к эквивалентной системе специального вида

 (2.21)

или, в векторной форме

x = ϕ(x),  (2.22)

где функции ϕ1(x), ϕ2(x), …, ϕn(x) - определены и непрерывны в некоторой окрестности искомого изолированного решения Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 21.03.19.png

Если выбрано некоторое начальное приближениеMacintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 21.03.29.png последующие приближения в методе простой итерации находятся по формулам (2.23)

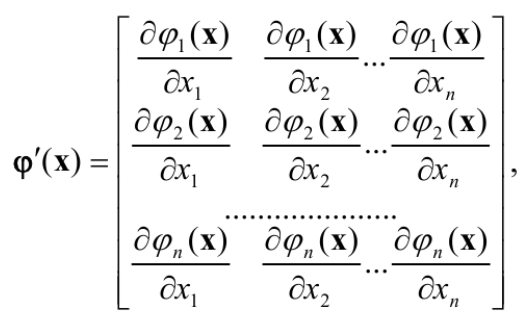
или, в векторной форме

x(k+1) = ϕ(x(k)). k = 0,1, 2,... (2.24)

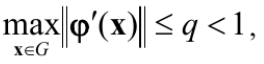
Если последовательность векторов Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 21.15.56.png сходится, то она сходится к решению Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 21.16.13.png

Достаточное условие сходимости итерационного процесса (2.23) формулируется следующим образом [3]:

**Теорема 2.4.**Пусть вектор-функция ϕ(x) непрерывна, вместе со своей производной



в ограниченной выпуклой замкнутой области G и

 (2.25)

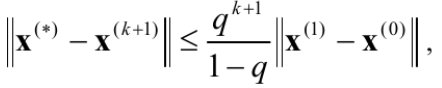
где q - постоянная. Если Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 21.22.48.pngи все последовательные приближения

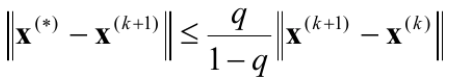
x(k+1) = ϕ(x(k)), k = 0,1, 2,...

также содержатся в G , то процесс итерации (2.23) сходится к единственному решению уравнения

x = ϕ(x)

в области G и справедливы оценки погрешности Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-21 в 21.22.53.png





Код программы

import numpy as np

import matplotlib.pyplot as plot

import math

# метод Ньютона

def Newton(x\_first):

epsilon = 0.01

x\_curr = x\_first

x\_next = []

i = 0

while True:

# объявляем функцию

x1, x2 = x\_curr

f\_x = np.array([x1 - math.cos(x2) - 2, x2 - math.sin(x1) - 2])

# матрица Якоби

J\_x = np.array([[1, math.sin(x2)], [-math.cos(x1), 1]])

x\_next = x\_curr + np.linalg.solve(J\_x, -1 \* f\_x)

# завершение

if norm(x\_next - x\_curr) <= epsilon:

break

x\_curr = x\_next

i += 1

return x\_next, i-1

def SimpleIterations(Phi):

epsilon = 0.01

x\_range = [(2.7, 2.9), (0.9, 1.1)]

x\_curr = np.array([[np.mean(x)] for x in x\_range])

x\_next = []

i = 0

u = [np.linspace(x[0], x[1]) for x in x\_range]

# максимум из матрицы Фи

q = max(map(lambda x: np.linalg.norm(Phi(x), ord=np.inf), zip(\*u)))

# коэфф

c = q / (1 - q)

while True:

# объявляем функцию

x1, x2 = x\_curr

x\_next = np.array([[2 + math.cos(x2)], [2 + math.sin(x1)]])

# завершение

if np.linalg.norm(x\_next - x\_curr, ord=np.inf) \* c <= epsilon:

break

x\_curr = x\_next

i += 1

return x\_next, i

def Phi\_matrix(x):

x1, x2 = x

return np.array([[0, -math.sin(x2)], [math.cos(x1), 0]])

u = np.arange(-2.0, 4.0, 0.1)

u1 = [(lambda x2: math.cos(x2) + 2)(x) for x in u]

u2 = [(lambda x1: math.sin(x1) + 2)(x) for x in u]

plot.figure()

plot.xlabel('x1')

plot.ylabel('x2')

plot.plot(u, u1, 'r')

plot.plot(u2, u,'g')

plot.axis('equal')

plot.grid(True)

plot.show()

x\_first = np.array([[2.8], [1.0]])

solve1, i1 = Newton(x\_first)

solve2, i2 = SimpleIterations(Phi\_matrix)

print('Newton: ',solve1,', iterations: ',i1)

print('SimpleIterations: ',solve2,', iterations:: ',i2)

Результат выполнения

Epsilon 0.01

Newton: [[1.03887718]

[2.86183545]] , iterations:: 9

SimpleIterations: [[1.03891629]

[2.86186928]] , iterations:: 10

Выводы

Метод Ньютона.

Использование метода Ньютона предполагает дифференцируемость функцийf1(x), f2(x),…, fn(x) и невырожденность матрицы Якоби (det J(x(k))≠ 0). В случае, если начальное приближение выбрано в достаточно малой окрестности искомого корня,итерации сходятся к точному решению, причем сходимость квадратичная.

Метод простой итерации.

Процесс итерации (2.23) сходится к единственному решению уравненияx = ϕ(x)

в области G.

Лабораторная работа №3

### 3.1 Интерполяция

## Задание

## 

11. , a) ; б) ; .

Теория

Пусть на отрезке [a, b] задано множество несовпадающих точек xi (интерполяционных узлов), в которых известны значения функции fi = f(xi),

i = 0, … ,n.

Приближающая функция ϕ(x, a) такая, что выполняются равенства

(3.1)

называется интерполяционной.

Наиболее часто в качестве приближающей функции используют многочлены степени n:

(3.2)

Подставляя в (3.2) значения узлов интерполяции и используя условие Pn(xi) = fi, получаем систему линейных алгебраических уравнений относительно коэффициентовai:

(3.3)

которая, в случае несовпадения узлов интерполяции имеет единственное решение.

Для нахождения интерполяционного многочлена не обязательно решать систему (3.3). Произвольный многочлен может быть записан в виде:

(3.4)

Здесь li(x) – многочлены степени n, так называемые лагранжевы многочлены влияния, которые удовлетворяют условиюи, соответственно, аинтерполяционный многочлен (3.4) запишется в виде

(3.5)

Интерполяционный многочлен, записанный в форме (3.5), называется ин- терполяционным многочленом Лагранжа.

Если ввести функцию, то выражение для интерполяционного многочлена Лагранжа примет вид:

(3.6)

Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются f(xi, xj) и определяются через разделенные разности нулевого порядка:



разделенные разности второго порядка определяются через разделенные разности первого порядка:



Разделенная разность порядка n - k + 2 определяется соотношениями

(3.7)

Таким образом, для (n+1)-й точки могут быть построены разделенные разности до n-го порядка; разделенные разности более высоких порядков равны нулю.

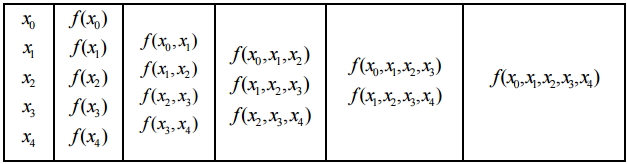
Пусть известны значения аппроксимируемой функции f(x) в точкахx0, x1, … , xn. Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции f(x) может быть записан в виде:

(3.8)

Запись многочлена в формуле (3.8) есть так называемый интерполяционный многочлен Ньютона. Если функция f(x) не есть многочлен n-й степени, то формула (3.8) для Pn(x) приближает функцию f(x) с некоторой погрешностью. Отметим, что при добавлении новых узлов первые члены многочлена Ньютона остаются неизменными.

Если функция задана в точках x0,x1, … , xn, то при построении интерполяционного многочлена Ньютона удобно пользоваться таблицей, называемой таблицей разделенных разностей, пример которой для n = 4 приведен в табл. 3.1.

**Таблица 3.1**



Для повышения точности интерполяции в сумму (3.8) 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.

Погрешность интерполяционных многочленов Лагранжа и Ньютона для случая аналитически заданной функции f(x) априорно может быть оценена по формуле, вывод которой приводится, например в [1].

(3.9)

где 

Если величину производных аппроксимируемой функции оценить сложно (например, для таблично заданной функции), то используется апостериорная оценка по первому отброшенному члену интерполяционного многочлена Ньютона, в который входят разделенные разности, являющиеся аналогами производных соответствующих порядков.

Код программы

import numpy as np

import math

def lagr(func, x2, x):

return np.sum( [func[i] \* np.prod([(x - x2[i]) / (x2[i] - x2[j]) if j != i else 1. for j in range(len(x2))]) for i in range(len(x2))])

def newt(func, x2, x):

n = len(x2)

coeff = [np.sum([func[k] / np.prod([(x2[k] - x2[j]) if j != k else 1. for j in range(i + 1)]) for k in range(i + 1)]) for i in range(n)]

return np.sum([coeff[i] \* np.prod([x - x2[j] for j in range(i)]) for i in range(n)])

x\_1 = [-3, -1, 1, 3]

x\_2 = [-3, 0, 1, 3]

M = 4.66

x\_val = -0.5

x2 = x\_1

func = [np.arctan(x) for x in x2]

print('Method :lagrange')

eps = np.abs(np.arctan(x\_val) - lagr(func, x2, x\_val))

print('arctg(x\*) = ',np.arctan(x\_val))

print('P(x)',poly(func, x2, x\_val))

print('|arctg(x\*) - P(x)| = ', eps)

upper\_bound = np.abs((x\_val - x2[0])\*(x\_val - x2[1])\*(x\_val - x2[2])\*(x\_val - x2[3])) \* M / math.factorial(len(x2) + 1)

x2 = x\_2

print('Method :newtone')

eps = np.abs(np.arctan(x\_val) - newt(func, x2, x\_val))

print('arctg(x\*) = ',np.arctan(x\_val))

print('P(x)',poly(func, x2, x\_val))

print('|arctg(x\*) - P(x)| = ', eps)

upper\_bound = np.abs((x\_val - x2[0])\*(x\_val - x2[1])\*(x\_val - x2[2])\*(x\_val - x2[3])) \* M / math.factorial(len(x2) + 1)

Результат выполнения

Method : lagrange

arctg(x\*) = 1.980790003

|arctg(x\*) - P(x)| = 0.0536429827643462

Method : newton

arctg(x\*) = 1.980789999134

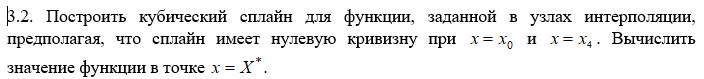
|arctg(x\*) - P(x)| = 0.0536493285865493

Выводы

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.

### 3.2 Кубический сплайн в узлах интерполяции

## Задание

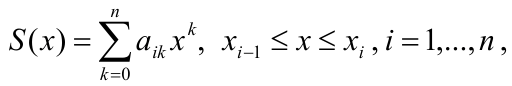


-0.5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -3.0 | -1.0 | 1.0 | 3.0 | 5.0 |
|  | 2.8198 | 2.3562 | 0.78540 | 0.32175 | 0.19740 |

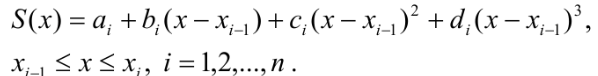
Теория

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

(3.10)

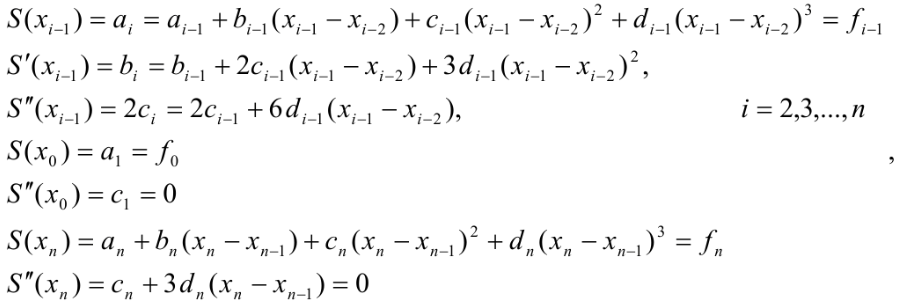
который в узлах интерполяции принимает значения аппрокимируемой функции и

непрерывен вместе со своими (n −1) производными. Такой кусочно-непрерывный

интерполяционный многочлен называется сплайном. Его коэффициенты находятся изусловий равенства в узлах сетки значений сплайна и приближаемой функции, а такжеравенства n −1 производных соответствующих многочленов. На практике наиболее частоиспользуется интерполяционный многочлен третьей степени, который удобнопредставить как (3.11)

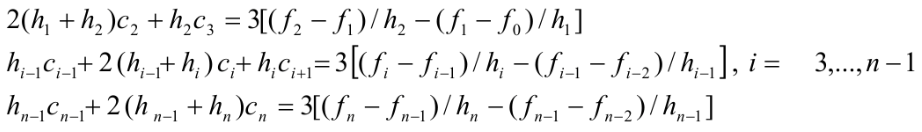
Для построения кубического сплайна необходимо построить n многочленов

третьей степени, т.е. определить 4n неизвестных ai,bi,ci,di. Эти коэффициенты ищутся изусловий в узлах сетки.

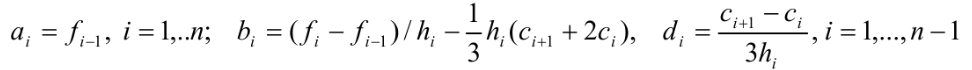
(3.12)

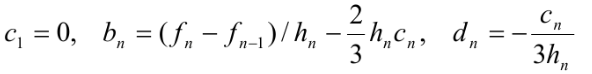
В (3.12) предполагается, что сплайны имеют нулевую кривизну на концах отрезка.В общем случае могут быть использованы и другие условия.

Если ввести обозначение hi = xi − xi−1, и исключить из системы (3.12) ai,bi, di, томожно получить систему из n −1 линейных алгебраических уравнений относительноci, i = 2,..., n с трехдиагональной матрицей:

(3.13)

Остальные коэффициенты сплайнов могут быть восстановлены по формулам:



(3.14)

Код программы

import numpy as np

import matplotlib.pyplot as plot

def TDMA(mat, D):

sz = len(mat)

x = Vector(sz)

p, q = [], []

p.append(-mat.c[0] / mat.b[0])

q.append(D[0] / mat.b[0])

for i in range(1, sz):

p\_i = 0 if i == sz - 1 else (-mat.c[i] / (mat.b[i] + mat.a[i] \* p[i - 1]))

q\_i = (D[i] - mat.a[i] \* q[i - 1]) / (mat.b[i] + mat.a[i] \* p[i - 1])

p.append(p\_i)

q.append(q\_i)

x[sz - 1] = q[sz - 1]

for i in range(sz - 2, -1, -1):

x[i] = p[i] \* x[i + 1] + q[i]

return x

def SPLINE(x, f):

n = len(x)

h = [x[i] - x[i - 1] for i in range(1, n)]

tridiag\_matrix = [[0, 2 \* (h[0] + h[1]), h[1]]]

b = [3 \* ((f[2] - f[1]) / h[1] - (f[1] - f[0]) / h[0])]

for i in range(1, n - 3):

tridiag\_row = [h[i], 2 \* (h[i] + h[i + 1]), h[i + 1]]

tridiag\_matrix.append(tridiag\_row)

b.append(3 \* ((f[i + 2] - f[i + 1]) / h[i + 1] - (f[i + 1] - f[i]) / h[i]))

tridiag\_matrix.append([h[-2], 2 \* (h[-2] + h[-1]), 0])

b.append(3 \* ((f[-1] - f[-2]) / h[-1] - (f[-2] - f[-3]) / h[-2]))

c = TDMA(tridiag\_matrix, b, n - 2)

a = []

b = []

d = []

c.insert(0, 0)

for i in range(1, n):

a.append(f[i - 1])

if i < n - 1:

d.append((c[i] - c[i - 1]) / (3 \* h[i - 1]))

b.append((f[i] - f[i - 1]) / h[i - 1] -

h[i - 1] \* (c[i] + 2 \* c[i - 1]) / 3)

b.append((f[-1] - f[-2]) / h[-1] - 2 \* h[-1] \* c[-1] / 3)

d.append(-c[-1] / (3 \* h[-1]))

return a, b, c, d

def polyval(x0, x, k, coef):

a, b, c, d = coef

diff = x0 - x[k]

return a[k] + b[k] \* diff + c[k] \* diff \*\* 2 + d[k] \* diff \*\* 3

def poly(x, x\_test, coef):

k = 0

for i, j in zip(x, x[1:]):

if i <= x\_test <= j:

break

k += 1

return polyval(x\_test, x, k, coef)

x = [-3, -1, 1, 3, 5]

f = [2.8198, 2.3562, 0.78540, 0.32175, 0.19740]

x\_test = -0.5

coef = SPLINE(x, f)

res = poly(x, x\_test, coef)

print(f'a = {coef[0]}\nb = {coef[1]}\nc = {coef[2]}\nd = {coef[3]}\n')

print('x\* = ', x\_test)

print('f(x\*) = ', res)

x\_vals = np.linspace(x[0], x[-1])

y = [poly(x, val, coef) for val in x\_vals]

plot.figure()

plot.xlabel('x')

plot.ylabel('y')

plot.plot(x\_vals, y, 'g')

plot.grid(True)

plot.show()

Результат выполнения

x = [-3.0, -1.0, 1.0, 3.0, 5.0]

y = [2.8198, 2.3562, 0.7854, 0.32175, 0.1974]

b = [-0.04700267857142859, -0.6013946428571427, -0.5990187499999999, -0.054205357142857145]

c = [0, -0.2771959821428571, 0.2783839285714286, -0.005977232142857136]

d = [-0.046199330357142854, 0.09259665178571429, -0.047393526785714284, 0.0009962053571428559]

x\* = -0.5

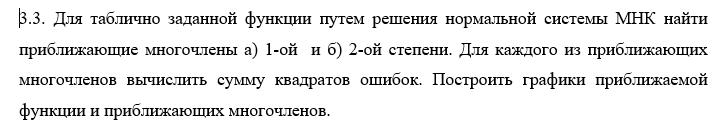
f(x\*) = 1.9977782645089286

Выводы

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.

### 3.3 Метод наименьших квадратов

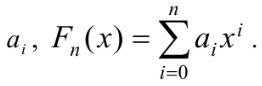
## Задание

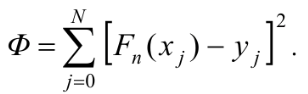


|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | -5.0 | -3.0 | -1.0 | 1.0 | 3.0 | 5.0 |
|  | 2.9442 | 2.8198 | 2.3562 | 0.7854 | 0.32175 | 0.1974 |

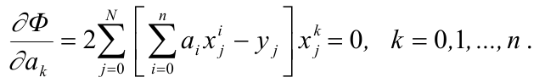
Теория

Пусть задана таблично в узлах xj функция yj = f (xj) , j = 0,1, ...,N . При этом

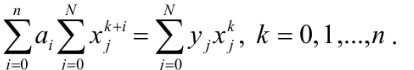
значения функции yj определены с некоторой погрешностью, также из физическихсоображений известен вид функции, которой должны приближенно удовлетворятьтабличные точки, например: многочлен степени n , у которого неизвестны коэффициентыНеизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции.

(3.15)

Минимума Φ можно добиться только за счет изменения коэффициентовмногочлена Fn(x) . Необходимые условия экстремума имеют вид

(3.16)

Эту систему для удобства преобразуют к следующему виду:

(3.17)

Система (3.17) называется нормальной системой метода наименьших квадратов(МНК) представляет собой систему линейных алгебраических уравнений относительнокоэффициентовai. Решив систему, построим многочлен

Fn(x) , приближающий функцию f(x) и минимизирующий квадратичное отклонение.

Код программы

import numpy as np

import matplotlib.pyplot as plt

from numpy.linalg import inv

x = [-5.0, -3.0, -1.0, 1.0, 3.0, 5.0]

y = [2.9442, 2.8198, 2.3562, 0.7854, 0.32175, 0.1974]

x = np.array(x)

t = np.array([[i \*\* j for i in x] for j in reversed(range(2))])

t\_tr = np.transpose(t)

g = np.dot(t, t\_tr)

second = np.dot(inv(g), np.dot(t,y))

print(second)#коэффы

x = np.array(x)

t = np.array([[i \*\* j for i in x] for j in reversed(range(3))])

t\_tr = np.transpose(t)

g = np.dot(t, t\_tr)

third = np.dot(inv(g), np.dot(t,y))

print(third)# -||-

x\_vals = np.linspace(x[0], x[-1])

y\_sec = [np.polyval(second, i) for i in x\_vals]

y\_thrd = [np.polyval(third, i) for i in x\_vals]

plt.scatter(x, y, color='r')

plt.plot(x\_vals, y\_sec, color='b')

plt.plot(x\_vals, y\_thrd, color='y')

plt.show()

y\_err = [np.polyval(second, i) for i in x]

err = sum([(y\_err[idx] - i) \*\* 2 for idx, i in enumerate(y)])

print('error 1 = ', err)

y\_err = [np.polyval(third, i) for i in x]

err = sum([(y\_err[idx] - i) \*\* 2 for idx, i in enumerate(y)])

print('error 2 =', err)

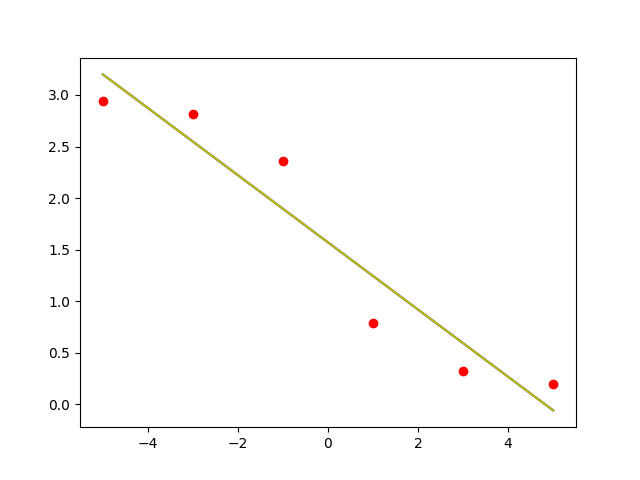
Результат выполнения

[-0.32569929 1.57079167]

[ 0.0 -0.325699286 1.57078906]

error 1 = 0.700686612047619

error 2 = 0.700686612017857



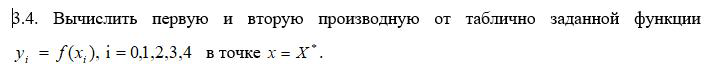
Приближающие многочлены 1 и 2 степени совпали.

Выводы

Система (3.17) с увеличением степени n приближающего многочлена становится плохо обусловленной и решение её связано с большой потерей точности. Поэтому при использовании метода наименьших квадратов, как правило,используют приближающий многочлен не выше третьей степени.

### 3.4 Численное дифференцирование

Задание



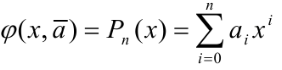
11. 1.0

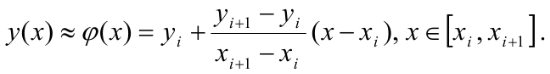
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 0.0 | 0.5 | 1.0 | 1.5 | 2.0 |
|  | 1.0 | 1.3776 | 1.5403 | 1.5707 | 1.5839 |

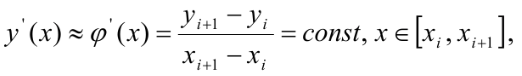
Теория

Формулы численного дифференцирования в основном используются при

нахождении производных от функции y = f (x) , заданной таблично. Исходная функцияyi = f(xi), i = 0,1...M на отрезках [xj, xj+k] заменяется некоторой приближающей, легко вычисляемой функциейMacintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.10.35.pngгде

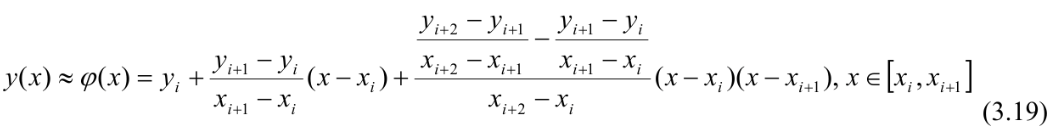
R(x)– остаточный член приближения, Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.11.24.png - набор коэффициентов, вообще говоря, различный для каждого из рассматриваемых отрезков, и полагают, что Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.12.18.png. Наиболее часто в качестве приближающей функцииMacintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.13.10.pngберется интерполяционный многочлен, а производные соответствующих порядков определяются дифференцированием многочлена.

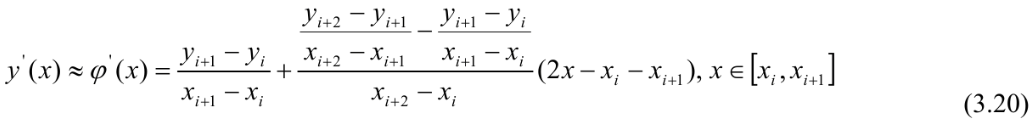
В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямойВ этом случае:

(3.18)

производная является кусочно-постоянной функцией и рассчитывается, по формуле (3.18)с первым порядком точности в крайних точках интервала, и со вторым порядком точности в средней точке интервала [1].

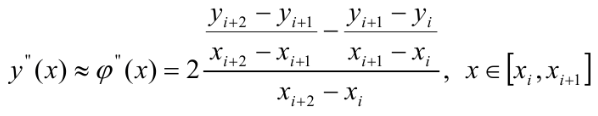
При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:





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

(3.21)

Код программы

x = [0.0, 0.5, 1.0, 1.5, 2.0]

y = [1.0, 1.3776, 1.5403, 1.5707, 1.5839]

x0 = 1.0

k = 0

for i, j in zip(x, x[1:]):

if i <= x0 <= j:

break

k += 1

result1 = (y[k + 1] - y[k]) / (x[k + 1] - x[k])

result2 = (y[k + 2] - y[k + 1]) / (x[k + 2] - x[k + 1]) - result1

result2 = result2 / (x[k + 2] - x[k])

ans1 = result1 + result2 \* (2 \* x0 - x[k] - x[k + 1])

result1 = (y[k + 2] - y[k + 1]) / (x[k + 2] - x[k + 1])

result2 = (y[k + 1] - y[k]) / (x[k + 1] - x[k])

ans2 = 2 \* (result1 - result2) / (x[k + 2] - x[k])

print('Первая производная', ans1)

print('Вторая производная', ans2)

Результат выполнения

Первая производная 0.19310000000000005

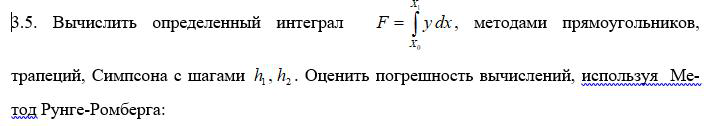
Вторая производная -0.5292000000000003

Выводы

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

### 3.5 Численное интегрирование

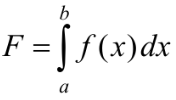
Задание

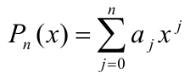


, ;

Теория

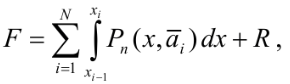
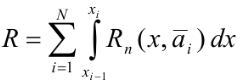
Формулы численного интегрирования используются в тех случаях, когда

вычислить аналитически определенный интеграл  не удается. Отрезок[a, b] разбивают точками x0 ....xN, так что a = x0 ≤ x1 ≤ ... ≤ xN = b с достаточно мелким шагом hi = xi − xi−1 и на одном или нескольких отрезках hi подынтегральную функцию f (x) заменяют такой приближающей ϕ(x) , так что она, во-первых, близка f (x) , а, во-вторых, интеграл от ϕ(x) легко вычисляется. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен

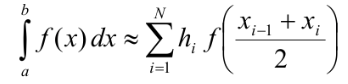
, причем коэффициенты многочлена aj, вообще говоря,

различны на каждом отрезке[xi, xi+k]и определяются из условия ϕ (xj) = f (xj), j = i,..., i + k , т.е. многочлен Pn зависит от параметров aj -Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.39.59.png, тогда

Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.40.40.png(3.22)

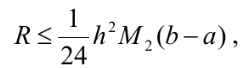
где Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.41.32.png– остаточный член интерполяции. Тогда где – остаточный член формулы численного интегрирования или её погрешность.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точкуMacintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.44.54.png, получим формулу прямоугольников.

(3.23)

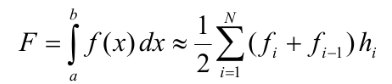
В случае постоянного шага интегрирования hi = h, i = 1,2,..., N и существованияимеет место оценка остаточного члена формулы

прямоугольников

(3.24)

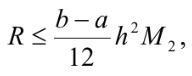
где M2 = max |f ′′(x)|[a,b].

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

(3.25)

Эта формула носит название формулы трапеций.

В случае постоянного шага интегрирования величина остаточного члена оценивается

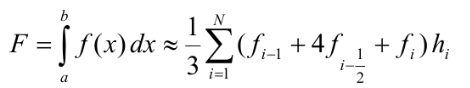
(3.26)

где M2 = max |f ′′(x)|[a,b].

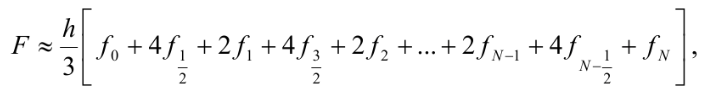
Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени,выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования:

Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.57.02.png

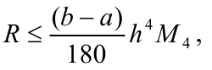
Для случаяMacintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.57.11.png, получим формулу Симпсона (парабол)

(3.27)

В случае постоянного шага интегрирования hi = h, i = 1,2,..., N , формула Симпсона принимает вид.

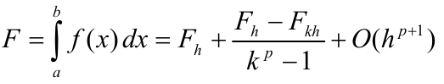
(3.28)

при этом количество интервалов на которое делится отрезок интегрирования, равно 2N. В том случае если существуетMacintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 3.59.58.pngдля оценки величины погрешности справедлива мажорантная оценка

(3.29)

где M4 = max |fIV(x)|[a ,b] .

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла на сетке с шагом h - F = Fh + O(hp) и на сетке с шагом kh - F = Fkh + O((kh)p), то

(3.30)

Код программы

iimport numpy as np

def Rectangle\_method(interval, f, h):

x = np.linspace(interval[0], interval[1], int((interval[1] - interval[0]) / h + 1))

integral = h \* np.sum([f((i + j) / 2) for i, j in zip(x, x[1:])])

return integral

def Simpson\_method(interval, f, size, h):

n = int((interval[1] - interval[0]) / h + 1)

integral = h \* (f(interval[0]) + 4 \* np.sum([f(interval[0] + i \* h) for i in range(1, n, 2)]) + 2 \* np.sum([f(interval[0] + i \* h) for i in range(2, n - 1, 2)]) + f(interval[1])) / 3

return integral

def Trap\_method(interval, f, h):

x = np.linspace(interval[0], interval[1], int((interval[1] - interval[0]) / h + 1))

y = [f(i) for i in x]

integral = h \* sum([i + j for i, j in zip(y[1:], y)]) / 2

return integral

def runge\_romberg(Ih, I2h, r, p):

return (Ih - I2h) / (1 - r \*\* p)

interval = [-2.0,2.0]

Hlist = [1.0,0.5]

r = Hlist[-1] / Hlist[0]

n = 2

func = lambda x: 1 / (x \*\* 3 + 64)

rect = []

trap = []

simp = []

for h in Hlist:

Rectangle\_method\_val = Rectangle\_method(interval, func, h)

rect.append(Rectangle\_method\_val)

Trap\_method\_val = Trap\_method(interval, func, h)

trap.append(Trap\_method\_val)

Simpson\_method\_val = Simpson\_method(interval, func, n, h)

simp.append(Simpson\_method\_val)

print('ответ \t\t\t\t\t\t\t\t\t\t погрешность')

print('метод прямоугольник:')

print(rect, runge\_romberg(rect[0], rect[-1], r, n))

print("метод трапеции:")

print(trap, runge\_romberg(trap[0], trap[-1], r, n))

print("метод Симпсона:")

print(simp, runge\_romberg(simp[0], simp[-1], r, n))

Результат выполнения

ответ погрешность

метод прямоугольник:

[0.0625872651271553, 0.06262558831675694] -0.00005109758613551

метод трапеции:

[0.06275564713064713, 0.06267145612890122] 0.00011225466899454177

метод Симпсона:

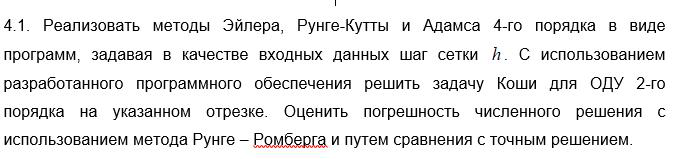
[0.06267551892551892, 0.06264339246165258] -0.00005109758613551

Выводы

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

### 4.1 Численные методы решения задачи Коши

Задание



|  |  |  |
| --- | --- | --- |
| 11 | , |  |

Теория:

Рассматривается задача Коши для одного дифференциального уравнения первого порядка

разрешенного относительно производной

y'= f (x, y)

y(x0 ) = y0 (4.1)

Требуется найти решение на отрезке [a,b], где x0 = a .

Введем разностную сетку на отрезке [a,b] Ω(k) = {xk = x0 + hk}, k = 0,1,...., N ,

h = |b – a| / N .

Точки xk - называются узлами разностной сетки, расстояния между узлами – шагом разностной сетки (h) , а совокупность значений какой либо величины заданных в узлах сетки называется сеточной функцией y(h) = {yk , k = 0,1,...., N}.

Приближенное решение задачи Коши (4.1) будем искать численно в виде сеточной функции y(h). Для оценки погрешности приближенного численного решения y(h) будем рассматривать это решение как элемент N+1- мерного линейного векторного пространства с какой либо нормой. В качестве погрешности решения принимается норма элемента этого пространства δ(h) = y(h) − [y](h) , где [y](h) - точное решение задачи (1) в узлах расчетной сетки. Таким образом εh = ||δ(h)||.

Метод Эйлера (явный)

Рассмотрим вывод соотношений метода Эйлера геометрическим способом.

Решение в узле x0 известно из начальных условий рассмотрим процедуру получения решения в узле x1 рис.4.1.

График функции y(h), которая является решением задачи Коши (1), представляет собой гладкую кривую, проходящую через точку (x0, y0) согласно условию y(x0) = y0, иимеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точкеx0и равен значению правой части дифференциального уравнения в точке (x0, y0) согласно выражению y' (x0) = f (x0, y0). Вслучае небольшого шага разностной сетки h график функции и график касательной неуспевают сильно разойтись друг от друга и можно в качестве значения решения в узле x1принять значение касательной y1, вместо значения неизвестного точного решения y1ист.При этом допускается погрешность |y1 − y1ист| геометрически представленная отрезком CD на рис.4.1. Из прямоугольного треугольника ABC находим СВ=ВА tg(CAB ) или Δy = hy' (x0). Учитывая, что Δy = y1 − y0 и заменяя производную y' (x0) на правую часть

дифференциального уравнения, получаем соотношение y1= y0 + hf (x0, y0). Считая теперь точку (x1, y1) начальной и повторяя все предыдущие рассуждения, получим

значение y2 в узле x2.

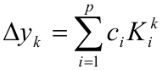
Переход к произвольным индексам дает формулу метода Эйлера:

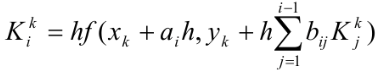
yk+1 = yk + hf (xk, yk) (4.2)

Методы Рунге-Кутты

Семейство явных методов Рунге-Кутты р-го порядка записывается в виде совокупности формул:

Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 0.52.30.png



(4.8)

Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 0.53.11.png

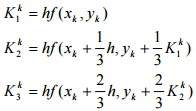
Метод Рунге-Кутты третьего порядка точности

Один из методов Рунге-Кутты третьего порядка

имеет вид:



(4.9)



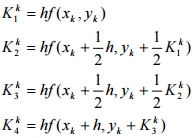
Метод Рунге-Кутты четвертого порядка





является одним из самых широко используемых методов для решения Задачи Коши:

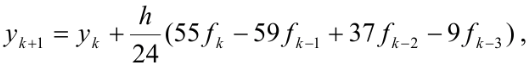
(4.10)



Метод Адамса 4-го порядка

При использовании интерполяционного многочлена 3-ей степени построенного по

значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

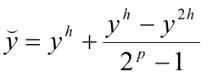
(4.25)

где fk значение подынтегральной функции в узле xk .

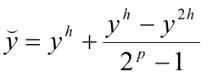
Контроль точности на каждом шаге h

Основным способом контроля точности получаемого численного решения при решении задачи Коши является методы основанные на принципе Рунге-Ромберга-Ричардсона.

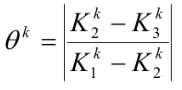
Пусть yh решение задачи Коши (1) полученоe методом Рунге-Кутты p – го порядка точности с шагом h в точке x+2h . Пусть y2h решение той же задачи в точке x+2h,полученное тем же методом, но с шагом 2h . Тогда выражение

(4.11)

аппроксимирует точное решение в точке x+2h y(x+2h) с p+1-ым порядком.

Второе слагаемое в выражении (4.11) оценивает главный член в погрешности решения yh , то есть . Контроль точности может быть организован следующим образом. Выбирается значение шага h и дважды рассчитывается решение вточке x+2h, один раз с шагом h, другой раз с шагом 2h. Рассчитывается величина Rh и сравнивается с заданной точностью ε . Если величина Rh меньше ε , то можнопродолжать вычисления с тем же шагом, в противном случае необходимо вернуться крешению в точке x, уменьшить шаг h и повторить вычисления.

Вычислительная стоимость такого контроля точности достаточно велика, особенно для многостадийных методов. Поэтому можно использовать более грубый способ контроля правильности выбора шага h . В случае метода Рунге-Кутты четвертого порядка точности следует на каждом шаге h рассчитывать параметр

(4.12)

Если величина θk порядка нескольких сотых единицы, то расчет продолжается стем же шагом, если θk больше одной десятой, то шаг следует уменьшить, если жеθk меньше одной сотой, то шаг можно увеличить.

Таким образом с помощью определения величин θk или Rh можно организовать

алгоритм выбора шага h для явного метода Рунге-Кутты.

Код программы

import numpy as np

def function(x, y, y1):

return y\*np.power(np.cos(x),2) - y1\*np.tan(x)

def original\_function(x):

return np.exp(np.sin(x))+ np.exp(-np.sin(x))

def runge\_kutta(f, xa, xb, ya, y1a, h):

n = int((xb - xa) / h)

x = xa

y = ya

z = y1a

res1 = [x]

res2 = [y]

res3 = [z]

for i in range(1, n + 1):

k1 = h \* z

l1 = h \* f(x, y, z)

k2 = h \* (z + 0.5 \* h \* f(x, y, z))

l2 = h \* f(x + 0.5 \* h, y + 0.5 \* k1, z + 0.5 \* h \* f(x, y, z))

k3 = h \* (z + 0.5 \* l2)

l3 = h \* f(x + 0.5 \* h, y + 0.5 \* k2, z + 0.5 \* l2)

k4 = h \* (z + l3)

l4 = h \* f(x + h, y + k3, z + l3)

x = xa + i \* h

y += (k1 + 2 \* k2 + 2 \* k3 + k4) / 6

z += (h \* f(x, y, z) + 2 \* l2 + 2 \* l3 + l4) / 6

res1.append(x)

res2.append(y)

res3.append(z)

return (res1, res2), res3

def euler(f, xa, xb, ya, y1a, h):

n = int((xb - xa) / h)

x = xa

y = ya

res1 = [x]

res2 = [y]

y1 = y1a

for i in range(n):

y1 += h \* f(x, y, y1)

y += h \* y1

x += h

res1.append(x)

res2.append(y)

return res1, res2

def adams(f, x, y, h, n, z):

z = z[:4] + [0] \* (len(z) - 4)

for i in range(3, n):

z[i + 1] = z[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]))

tmp = y[i] + h/24 \* (55\*z[i] - 59\*z[i - 1] + 37\*z[i - 2] - 9\*z[i - 3])

x.append(x[-1] + h)

y.append(tmp)

return x, y

h = 0.1

y = 2

y\_d = 0

a = 0

b = 1

euler\_r = euler(function, a, b, y, y\_d, h)

euler\_r\_half\_h = euler(function, a, b, y, y\_d, h / 2)

print('Euler')

table = PrettyTable(['X', 'Y', 'Точный Y', '|Погрешность|', 'Рунге Ромберг'])

for x, y, yr in zip(\*euler\_r, euler\_r\_half\_h[1]):

tmp = original\_function(x)

table.add\_row([round(x, 2), round(y, 3), round(tmp, 3), abs(y - tmp), abs(y - yr) / (2\*\*1 - 1)])

print(table)

runge\_r, z = runge\_kutta(function, a, b, y, y\_d, h)

runge\_r\_half\_h, z\_half\_h = runge\_kutta(function, a, b, y, y\_d, h / 2)

print('Runge-Kutta')

table = PrettyTable(['X', 'Y', 'Точный Y', '|Погрешность|', 'Рунге Ромберг'])

for x, y, yr in zip(\*runge\_r, runge\_r\_half\_h[1]):

tmp = original\_function(x)

table.add\_row([round(x, 2), round(y, 3), round(tmp, 3), abs(y - tmp), abs(y - yr) / (2\*\*4 - 1)])

print(table)

adams\_r = adams(function, runge\_r[0][:4], runge\_r[1][:4], h, int((b - a) / h), z)

adams\_r\_half\_h = adams(function, runge\_r[0][:4], runge\_r[1][:4], h / 2, int((b - a) / h), z\_half\_h)

print('Adams')

table = PrettyTable(['X', 'Y', 'Точный Y', '|Погрешность|', 'Рунге Ромберг'])

for x, y, yr in zip(\*adams\_r, adams\_r\_half\_h[1]):

tmp = original\_function(x)

table.add\_row([round(x, 2), round(y, 3), round(tmp, 3), abs(y - tmp), abs(y - yr) / (2\*\*4 - 1)])

print(table)

Результат выполнения

Эйлер

+-----+-------+------------+----------------------+----------------------+

| X | Y | Точный Y | Рунге Ромберг | Погрешность |

+-----+-------+------------+----------------------+----------------------+

| 0 | 2 | 2.0 | 0.0 | 0.0 |

| 0.1 | 2.02 | 2.01 | 0.010025008225857057 | 0.015000000000000124 |

| 0.2 | 2.08 | 2.04 | 0.020198509036243095 | 0.04481053432989146 |

| 0.3 | 2.169 | 2.088 | 0.030604627088965053 | 0.08866215398719213 |

| 0.4 | 2.295 | 2.154 | 0.041295126313502184 | 0.1451824867950866 |

| 0.5 | 2.447 | 2.234 | 0.05227084043683927 | 0.21237586095219285 |

| 0.6 | 2.259 | 2.327 | 0.06346307389279682 | 0.28761489518406025 |

| 0.7 | 2.504 | 2.43 | 0.07471595255830188 | 0.3676554908771621 |

| 0.8 | 2.971 | 2.537 | 0.07577114758695759 | 0.44868687270442686 |

| 0.9 | 3.169 | 2.646 | 0.0762567004827557 | 0.5264263375445783 |

| 1.0 | 3.341 | 2.751 | 0.08568168940806855 | 0.5962640681927227 |

+-----+-------+------------+----------------------+----------------------+

Рунге Кутта

+-----+-------+------------+----------------------+-------------------------+

| X | Y | Точный Y | Рунге Ромберг | Погрешность |

+-----+-------+------------+----------------------+-------------------------+

| 0 | 2 | 2.0 | 0.0 | 0.0 |

| 0.1 | 2.01 | 2.01 | 2.909279395169051e-08 | 4.984388694190045e-06 |

| 0.2 | 2.04 | 2.04 | 7.293203463376585e-08 | 1.974971603358546e-06 |

| 0.3 | 2.088 | 2.088 | 1.2582233166469337e-07 | 4.37309333490011e-06 |

| 0.4 | 2.154 | 2.154 | 1.7949052644183894e-07 | 7.598228869553244e-06 |

| 0.5 | 2.234 | 2.234 | 2.2474211025524937e-07 | 1.15175941391778e-06 |

| 0.6 | 2.327 | 2.327 | 2.5396732583615744e-07 | 1.596085721531352e-06 |

| 0.7 | 2.43 | 2.43 | 2.645045742433183e-07 | 2.07222964568639e-06 |

| 0.8 | 2.537 | 2.537 | 2.625982937765059e-07 | 2.556513605891754e-06 |

| 0.9 | 2.646 | 2.646 | 2.674252823631207e-07 | 3.02285902420799e-06 |

| 1.0 | 2.751 | 2.751 | 3.1468522987410097e-07 | 3.44378549706039e-07 |

+-----+-------+------------+------------------------+-----------------------+

Адамс

+-----+-------+------------+----------------------+-------------------------+

| X | Y | Точный Y | Рунге Ромберг | Погрешность |

+-----+-------+------------+----------------------+-------------------------+

| 0 | 2 | 2.0 | 0.0 | 0.0 |

| 0.1 | 2.01 | 2.01 | 2.909279395169051e-08 | 0.0004984388694190045|

| 0.2 | 2.04 | 2.04 | 7.293203463376585e-08 | 0.0019749716033358546|

| 0.3 | 2.088 | 2.088 | 1.2582233166469337e-07 | 0.004373093334900110 |

| 0.4 | 2.154 | 2.154 | 1.7301417454262946e-06 | 0.0075983289455600735|

| 0.5 | 2.234 | 2.234 | 7.327568085635505e-07 | 0.001517519483380194 |

| 0.6 | 2.327 | 2.327 | 1.578112386502184e-05 | 0.005959772550869496 |

| 0.7 | 2.43 | 2.43 | 4.865540120402301e-05 | 0.00201901676220058 |

| 0.8 | 2.537 | 2.537 | 0.00010578355417445451 | 0.0025580517305748 |

| 0.9 | 2.645 | 2.646 | 0.00018998651413548728 | 0.003015905751602476 |

| 1.0 | 2.751 | 2.751 | 0.0003001909174225048 | 0.003443847819150686 |

+-----+-------+------------+------------------------+-----------------------+

Выводы

Метод Эйлера (явный)

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 вряд Тейлора, конечно разностным методом (с помощью разностной аппроксимации производной), квадратурным способом (использованием эквивалентного интегрального уравнения).

Методы Рунге-Кутты

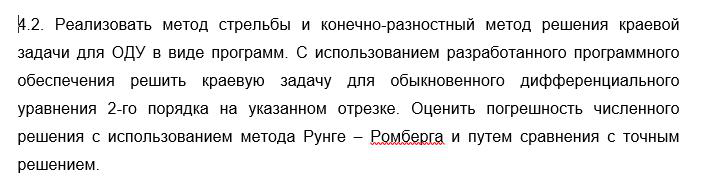
Параметры ai,bij, ci, в методе Рунге-Кутты подбираются так, чтобы значение yk+1, рассчитанное по соотношению (4.8) совпадало со значением разложения в точке xk+1 точного решения вряд Тейлора с погрешностью O(hp+1).

Метод Адамса 4-го порядка

Метод Адамса(4.25) 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 x0 решение y0 известно из начальных условий,а в других трех узлахx1, x2, x3 решенияy1, y2, y3 можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка(4.10).

### 4.2 Численные методы решения краевой задачи для ОДУ

## Задание



|  |  |
| --- | --- |
| x(x-1)y-xy+ y=0,  y(1)=2,  2y(2)-y(2)=1 | y(x)=1+x+x ln|x| |

Теория:

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

y' '= f (x, y, y') (4.28)

с граничными условиями, заданными на концах отрезка [a,b].

y(a) = y0

y(b) = y1 (4.29)

Следует найти такое решение y(x) на этом отрезке, которое принимает на концах

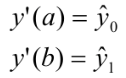
отрезка значения y0 , y1 . Если функция f (x, y, y') линейна по аргументам y , y' , то задача

(4.28),(4.29) - линейная краевая задача, в противном случае – нелинейная.

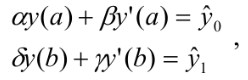
Кроме граничных условий (4.29) называемых граничными условиями первого рода,

используются еще условия на производные от решения на концах - граничные условия

второго рода:

(4.30)

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

(4.31)

где α ,β ,δ ,γ - такие числа, что |α| + |β| ≠ 0, |δ| + |γ| ≠ 0 .

Возможно на разных концах отрезка использовать условия различных типов. Далее рассматриваются два приближенных метода решения краевой задачи:

* метод стрельбы (пристрелки);
* конечно-разностный метод.

Метод стрельбы

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

Пусть надо решить краевую задачу (4.28), (4.29) на отрезке [a,b]. Вместо

исходной задачи формулируется задача Коши с уравнением (4.28) и с начальными условиями

y(a) = y0 ,

y' (b) =η (4.32)

где η - некоторое значение тангенса угла наклона касательной к решению в точке x = a. Положим сначала некоторое начальное значение параметру η =η0, после чего решим каким либо методом задачу Коши (4.28),(4.32). Пусть y = y0 (x, y0, η0) решение этой задачи на интервале [a,b], тогда сравнивая значение функции y0 (b, y0, η0) со значениемy1 в правом конце отрезка можно получить информацию для корректировки угла наклона касательной к решению в левом конце отрезка. Решая задачу Коши для нового значения η =η1, получим другое решение со значением y1 (b, y0, η1) на правом конце. Таким образом, значение решения на правом конце y(b, y0, η) будет являться функцией одной переменной η. Задачу можно сформулировать таким образом: требуется найти такое значение переменной η\*, чтобы решение y(b, y0, η\*) в правом конце отрезка совпало со значением y1 из (4.29). Другими словами решение исходной задачи эквивалентно нахождению корня уравнения

Φ(η) = 0,

где Φ(η) = y(b, y0, η) − y1. (4.33)

Конечно-разностный метод решения краевой задачи

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке [a,b]

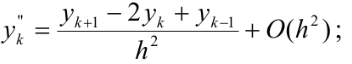
y' '+ p(x) y'+q(x) y = f (x) (4.35)

y(a) = y0 , y(b) = y1. (4.36)

Введем разностную сеткунаотрезке [a,b] Ω(h) = {xk = x0 + hk}, k = 0,1,...., N ,

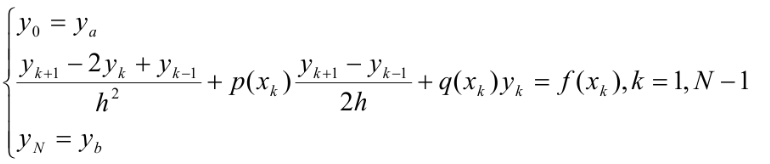
h = |b – a| / N . Решение задачи (4.35),(4.36) будем искать в виде сеточной функции y(h ) = {yk, k = 0,1,...., N}, предлагая, что решение существует и единственно. Введе мразностную аппроксимацию производных следующим образом:

Macintosh HD:Users:vladislava:Desktop:Снимок экрана 2017-05-22 в 0.17.36.png

(4.37)

Подставляя аппроксимации производных из (4.37) в (4.35),(4.36) получим систему

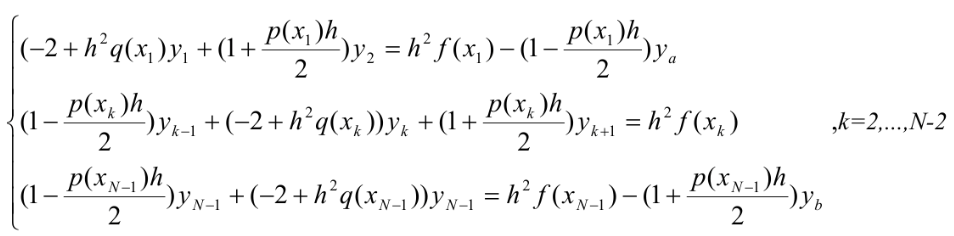
уравнений для нахождения yk:

(4.38)

Приводя подобные и учитывая, что при задании граничных условий первого рода

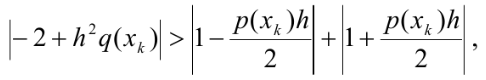
два неизвестныхy0, yN уже фактически определены, получим систему линейных

алгебраических уравнений с трехдиагональной матрицей коэффициентов

(4.39)

Для системы (4.39) при достаточно малых шагах сетки h и q(xk) < 0 выполнены

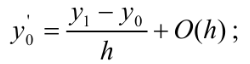
условия преобладания диагональных элементов

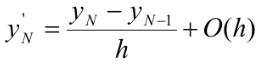
(4.39)

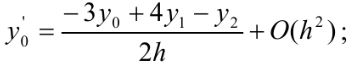
что гарантирует устойчивость счета и корректность применения метода прогонки для решения этой системы.

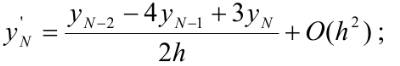
В случае использования граничных условий второго и третьего рода

аппроксимация производных проводится с помощью односторонних разностей первого и второго порядков.



(4.40)



(4.41)

Код программы

import numpy as np

from tdma import TDMA

from l41 import RK

def function(x, y, y1):

return (x\*y1-y)/(x\*(x-1))

def original\_function(x):

return 1+x+x\*np.log(np.abs(x))

def p(x):

return -1/(x-1)

def q(x):

return 1/(x\*\*2-x)

def f(x):

return 0

def deriv(xi, yi, x):

i = 0

while xi[i + 1] < x - 1e-7:

i += 1

return (yi[i + 1] - yi[i]) / (xi[i + 1] - xi[i])

def N\_function(cur\_n, prev\_n, ans\_cur, ans\_prev, alpha1, beta1, B, b):

n1 = beta1 \* deriv(ans\_cur[0], ans\_cur[1], b)

n2 = beta1 \* deriv(ans\_prev[0], ans\_prev[1], b)

n3 = alpha1 \* ans\_prev[1][len(ans\_prev[0]) - 1]

n4 = alpha1 \* ans\_cur[1][len(ans\_cur[0]) - 1] + n1 - B

n5 = alpha1 \* ans\_cur[1][len(ans\_cur[0]) - 1] + n1 - n3 - n2

return cur\_n - n4 \* (cur\_n - prev\_n) / n5

def Shooting(a, b, h, eps, f, alpha0, alpha1, beta0, beta1, A, B):

n\_prev = 1

n\_cur = 0.8

ans\_prev = RK(f, a, b, n\_prev, (A - alpha0 \* n\_prev) / beta0, h)[0]

ans\_cur = RK(f, a, b, n\_cur, (A - alpha0 \* n\_cur) / beta0, h)[0]

while abs(alpha1 \* ans\_cur[1][len(ans\_cur[0]) - 1] + \

beta1 \* deriv(ans\_cur[0], ans\_cur[1], b) - B) > eps:

n = N\_function(n\_cur, n\_prev, ans\_cur, ans\_prev, alpha1, beta1, B, b)

n\_prev = n\_cur

n\_cur = n

ans\_prev = ans\_cur

ans\_cur = RK(f, a, b, n\_cur, (A - alpha0\*n\_cur) / beta0, h)[0]

return ans\_cur

def finite\_difference\_method(a1, b1, h, alpha\_0, alpha\_1, beta\_0, beta\_1, A, B):

x = [a1]

a = []

b = []

c = []

d = []

n = round((b1 - a1) / h)

a.append(0)

b.append(-2 / (h \* (2 - p(a1) \* h)) + q(a1) \* h / (2 - p(a1) \* h) + alpha\_0 / beta\_0)

c.append(2 / (h \* (2 - p(a1) \* h)))

d.append(A / beta\_0 + h \* f(a1) / (2 - p(a1) \* h))

x.append(x[0] + h)

for i in range(1, n):

a.append(1 / h\*\*2 - p(x[i]) / (2 \* h))

b.append(-2 / h\*\*2 + q(x[i]))

c.append(1 / h\*\*2 + p(x[i]) / (2 \* h))

d.append(f(x[i]))

x.append(x[i] + h)

a.append(-2 / (h \* (2 + p(x[n]) \* h)))

b.append(2 / (h \* (2 + p(x[n]) \* h)) - q(x[n]) \* h /

(2 + p(x[n]) \* h) + alpha\_1 / beta\_1)

c.append(0)

d.append(B / beta\_1 - h \* f(x[n]) / (2 + p(x[n]) \* h))

y = TDMA(a, b, c, d, len(a))

return x, y

a = 2

b = 3

alpha0 = 0

alpha1 = 1

beta0 = 1

beta1 = 1

y0 = 2+np.log(2)

y10 = 6+4\*np.log(3)

step = 0.1

eps = 0.00001

res1 = Shooting(a, b, step, eps, function, alpha0, alpha1, beta0, beta1, y0, y10)

res2 = Shooting(a, b, step / 2, eps, function, alpha0, alpha1, beta0, beta1, y0, y10)

print('Метод стрельбы')

answer\_t = tab(['X', 'Y', 'Точное Y', '|Y - Точное Y|', 'Рунге Ромберг'])

for x, y, yr in zip(\*res1,res2[1]):

tmp = original\_function(x)

answer\_t.add\_row([round(x, 2), y, tmp, abs(y - tmp), abs(y - yr) / (2\*\*2 - 1)])

print(answer\_t)

res1 = finite\_difference\_method(a, b, step, alpha0, alpha1, beta0, beta1, y0, y10)

res2 = finite\_difference\_method(a, b, step / 2, alpha0, alpha1, beta0, beta1, y0, y10)

print('Конечных разностей')

answer\_t = tab(['X', 'Y', 'Точное Y', '|Y - Точное Y|', 'Рунге Ромберг'])

for x, y, yr in zip(\*res1, res2[1]):

tmp = original\_function(x)

answer\_t.add\_row([round(x, 2), y, tmp, abs(y - tmp), abs(y - yr) / (2\*\*1 - 1)])

print(answer\_t)

Результат выполнения

Метод стрельбы

+-----+--------------------+--------------------+----------------------+----------------------+

| X | Y | Точное Y | |Y - Точное Y| | Рунге Ромберг |

+-----+--------------------+--------------------+----------------------+----------------------+

| 2 | 4.428964033056604 | 4.386294361119891 | 0.04266967193671345 | 0.006996360495046853 |

| 2.1 | 4.700633205188061 | 4.6580684239316925 | 0.042564781256368533 | 0.052464825322667906 |

| 2.2 | 4.976862812486561 | 4.934606192801395 | 0.04225661968516636 | 0.09905570498410121 |

| 2.3 | 5.257445417444974 | 5.215690982750738 | 0.04175443469423623 | 0.14670932521756175 |

| 2.4 | 5.54219163812136 | 5.501124969649359 | 0.04106666847200113 | 0.19537158962342924 |

| 2.5 | 5.83092788850302 | 5.790726829685388 | 0.04020105881763136 | 0.24499325650780435 |

| 2.6 | 6.1234944809026475 | 6.084329757071335 | 0.03916472383131264 | 0.2955293337307108 |

| 2.7 | 6.41974402066386 | 6.3817797871277655 | 0.03796423353609413 | 0.34693856860868255 |

| 2.8 | 6.719540038970009 | 6.682934368107242 | 0.03660567086276689 | 0.39918301505657006 |

| 2.9 | 7.022755821182356 | 6.9876611372780415 | 0.03509468390431447 | 0.4522276639990759 |

| 3.0 | 7.32927339695419 | 7.295836866004329 | 0.03343653094986099 | 0.5060401259955277 |

+-----+--------------------+--------------------+----------------------+----------------------+

Конечных разностей

+-----+--------------------+--------------------+-----------------------+-----------------------+

| X | Y | Точное Y | |Y - Точное Y| | Рунге Ромберг |

+-----+--------------------+--------------------+-----------------------+-----------------------+

| 2 | 4.385068985236712 | 4.386294361119891 | 0.0012253758831786143 | 0.0009192303866027984 |

| 2.1 | 4.656886766732415 | 4.6580684239316925 | 0.0011816571992770974 | 0.13561600074016766 |

| 2.2 | 4.933472288113276 | 4.934606192801395 | 0.00113390468811847 | 0.27569908620880756 |

| 2.3 | 5.2146088339299705 | 5.21569098275074 | 0.0010821488207692909 | 0.41914235546388046 |

| 2.4 | 5.500098533554851 | 5.501124969649361 | 0.0010264360945093642 | 0.5657756318463294 |

| 2.5 | 5.789760005579247 | 5.790726829685389 | 0.000966824106142461 | 0.7154439690993195 |

| 2.6 | 6.083426379107174 | 6.084329757071336 | 0.0009033779641614004 | 0.8680057565830692 |

| 2.7 | 6.380943619465422 | 6.381779787127767 | 0.0008361676623449554 | 1.0233311216085559 |

| 2.8 | 6.68216910195657 | 6.6829343681072455 | 0.0007652661506751812 | 1.1813005732930097 |

| 2.9 | 6.986970389361586 | 6.987661137278045 | 0.0006907479164590669 | 1.3418038443231053 |

| 3.0 | 7.29522417806275 | 7.295836866004332 | 0.0006126879415822017 | 1.5047388960927215 |

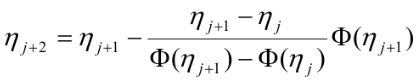
+-----+--------------------+--------------------+-----------------------+-----------------------+

Выводы

Метод стрельбы

Уравнение (4.33) является “алгоритмическим” уравнением, так как левая часть егозадается с помощью алгоритма численного решения соответствующей задачи Коши. Но методы решения уравнения (4.33) аналогичны методам решения нелинейных уравнений, изложенным в разделе о решении нелинейных уравнений и систем нелинейных уравнений. Следует заметить, что так как невозможно вычислитьпроизводную функции Φ(η) , то вместо метода Ньютона следует использовать методсекущих, в котором производная от функции заменена ее разностным аналогом. Данныйразностный аналог легко вычисляется по двум приближениям, например ηk и ηk+1.

Следующее значение искомого корня определяется по соотношению

(4.34)

Итерации по формуле (4.34) выполняются до удовлетворения заданной точности.

Конечно-разностный метод решения краевой задачи

В случае использования формул(4.40) линейная алгебраическая система

аппроксимирует дифференциальную задачу в целом только с первым порядком (из-зааппроксимации в граничных точках), однако сохраняется трех диагональная структураматрицы коэффициентов. В случае использования формул (4.41) второй порядокаппроксимации сохраняется везде, но матрица линейной системы не трехдиагональная.