МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторные работы**

**по курсу «Численные методы»**

Выполнил: А.С. Федоров

Преподаватель: Д.Л. Ревизников

Группа: 8О-407Б Дата:

Оценка:

Подпись:

Москва, 2022

1. **Численные методы линейной алгебры**
   1. **LU-разложение матриц**

**Задача**

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

**Вариант 23**



**Алгоритм решения**

LU – разложение матрицы A представляет собой разложение матрицы A в  
произведение нижней и верхней треугольных матриц, т.е.  
*A = LU,*   
где L - нижняя треугольная матрица (матрица, у которой все элементы, находящиеся  
выше главной диагонали равны нулю), U - верхняя треугольная матрица  
(у которой все элементы, находящиеся ниже главной диагонали равны нулю).

LU – разложение может быть построено с использованием описанного выше

метода Гаусса. В методе Гаусса матрица СЛАУ с помощью равносильных преобразований преобразуется в верхнюю треугольную матрицу, получающуюся в результате прямого хода. В обратном ходе определяются неизвестные.

Так как исходный код для лабораторных работ раздела 1 был утерян при форматировании жесткого диска, будут приведены только алгоритмы решения задач, согласно которым были реализованы программы.

* 1. **Метод прогонки**

**Задача**

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

**Вариант 23**



**Алгоритм решения**

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. 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 .

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.

* 1. **Итерационные методы решения СЛАУ**

**Задача**

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

**Вариант 23**



**Алгоритм решения**

При большом числе уравнений прямые методы решения СЛАУ (за исключением

метода прогонки) становятся труднореализуемыми на ЭВМ прежде всего из-за сложности

хранения и обработки матриц большой размерности. В то же время характерной

особенностью ряда часто встречающихся в прикладных задачах СЛАУ является

разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с

их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее

использовать итерационные методы.

Методы последовательных приближений, в которых при вычислении последующего

приближения решения используются предыдущие, уже известные приближенные

решения, называются итерационными.

Идея метода заключается в преобразовании задачи в вид, в котором отображение аргумента функцией будет сжимающим. Последовательное применение таких сжимающих преобразований будет сходиться к решению. Количество итераций зависит от точности, которую требуется достичь.

Метод Зейделя отличается от метода простых итераций только тем, что при вычислении значений на текущей итерации используются уже вычисленные значение на этой же итерации в комбинации с предыдущей, где новые значений еще неизвестны. Это позволяет сократить число итераций для достижения аналогичной точности.

* 1. **Метод вращений**

**Задача**

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

**Вариант 23**



**Алгоритм решения**

Метод вращений Якоби применим только для симметрических матриц и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия Поскольку для симметрических матриц

матрица преобразования подобия U является ортогональной , то ,

где - диагональная матрица с собственными значениями на главной диагонали

* 1. **QR-алгоритм**

**Задача**

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

**Вариант 23**



**Алгоритм решения**

При решении полной проблемы собственных значений для несимметричных  
матриц эффективным является подход, основанный на приведении матриц к подобным,  
имеющим треугольный или квазитреугольный вид. Одним из наиболее распространенных  
методов этого класса является QR-алгоритм, позволяющий находить как вещественные,  
так и комплексные собственные значения.

В основе QR-алгоритма лежит представление матрицы в виде *A = QR*, где - *Q*

ортогональная матрица, а R - верхняя треугольная. Такое разложение

существует для любой квадратной матрицы. Одним из возможных подходов к построению

QR разложения является использование преобразования Хаусхолдера, позволяющего

обратить в нуль группу поддиагональных элементов столбца матрицы.

### 2 Численные методы решения нелинейных уравнений и систем.

**2.1 Решение нелинейных уравнений**

**Задача**

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

**Вариант 23**



**Метод Ньютона**

Исходный код

def f(x):

    return np.log(x+2)-x\*\*4+0.5

def Newton\_method(f, df, x0, e):

    counter = 0

    x = x0

    print('Итерация '+str(counter)+': x = '+str(x))

    counter += 1

    x\_next = x-(f(x)/df(x))

    print('Итерация '+str(counter)+': x = '+str(x\_next))

    counter += 1

    while abs(x\_next - x) >= e:

        x = x\_next

        x\_next = x-(f(x)/df(x))

        print('Итерация '+str(counter)+': x = '+str(x\_next))

        counter += 1

    return x\_next

def df(x):

    return 1/(x+2)-4\*x\*\*3

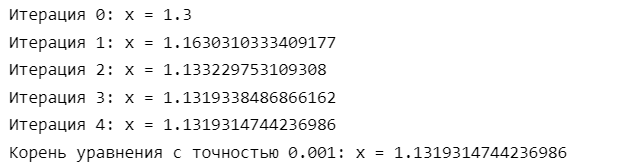
x0 = 1.3

e = 0.001

answer = Newton\_method(f, df, x0, e)

print('Корень уравнения с точностью '+str(e)+': x = '+str(answer))

Результат работы



**Метод простых итераций**

Исходный код

def phi(x):

    return np.power(np.log(x+2)+0.5, 1/4)

def dphi(x):

    return 1/((4\*x+8)\*np.power(np.log(x+2)+0.5, 3/4))

def Simple\_iteration\_method(phi, x0, q, e):

    counter = 0

    x = x0

    print('Итерация '+str(counter)+': x = '+str(x))

    counter += 1

    x\_next = phi(x)

    print('Итерация '+str(counter)+': x = '+str(x\_next))

    counter += 1

    while q/(1-q)\*abs(x\_next-x) > e:

        x = x\_next

        x\_next = phi(x)

        print('Итерация '+str(counter)+': x = '+str(x\_next))

        counter += 1

    return x\_next

x0 = 1.0

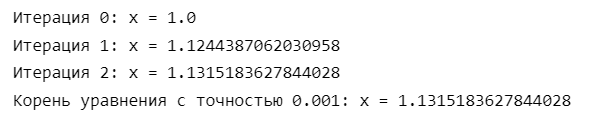
q = 0.0586154

e = 0.001

answer = Simple\_iteration\_method(phi, x0, q, e)

print('Корень уравнения с точностью '+str(e)+': x = '+str(answer))

Результат работы



**2.2 Решение систем нелинейных уравнений**

**Задача**

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

**Вариант 23**

, где *а = 2*.

**Метод Ньютона**

Исходный код

def f1(x):

    return 2\*x[0]\*\*2-x[0]+x[1]\*\*2-1

def f2(x):

    return x[1]-np.tan(x[0])

def df1x1(x):

    return 2\*2\*x[0]-1

def df1x2(x):

    return 2\*x[1]

def df2x1(x):

    return -1/(np.cos(x[0])\*\*2)

def df2x2(x):

    return 1

def f(x):

    return np.array([f1(x), f2(x)])

def J(x):

    return np.array([[df1x1(x), df1x2(x)],

                     [df2x1(x), df2x2(x)]])

def Newton\_method(f, J, x0, e):

    counter = 0

    x = x0

    print('Итерация '+str(counter)+': x = '+str(x))

    counter += 1

    delta\_x = np.linalg.solve(J(x), -f(x))

    x\_next = x+delta\_x

    print('Итерация '+str(counter)+': x = '+str(x\_next))

    counter += 1

    while np.linalg.norm(x\_next-x) > e:

        x = x\_next

        delta\_x = np.linalg.solve(J(x), -f(x))

        x\_next = x+delta\_x

        print('Итерация '+str(counter)+': x = '+str(x\_next))

        counter += 1

    return x\_next

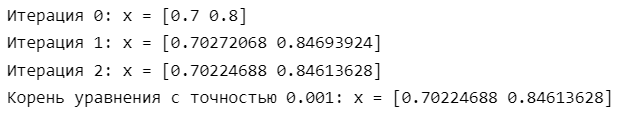
x0 = np.array([0.7, 0.8])

e = 0.001

answer = Newton\_method(f, J, x0, e)

print('Корень уравнения с точностью '+str(e)+': x = '+str(answer))

Результат работы



**Метод простых итераций**

Исходный код

def f1(x):

    return 2\*x[0]\*\*2-x[0]+x[1]\*\*2-1

def f2(x):

    return x[1]-np.tan(x[0])

x0 = np.array([1, 0.75])

def df1x1(x):

    return 2\*2\*x[0]-1

def df1x2(x):

    return 2\*x[1]

def df2x1(x):

    return -1/(np.cos(x[0])\*\*2)

def df2x2(x):

    return 1

def f(x):

    return np.array([f1(x), f2(x)])

def J(x):

    return np.array([[df1x1(x), df1x2(x)],

                     [df2x1(x), df2x2(x)]])

J\_inv = np.linalg.inv(J(x0))

def phi\_1(X):

    return X[0]-J\_inv[0,0]\*(2\*X[0]\*\*2-X[0]+X[1]\*\*2-1)-J\_inv[0,1]\*(X[1]-np.tan(X[0]))

def phi\_2(X):

    return X[1]-J\_inv[1,0]\*(2\*X[0]\*\*2-X[0]+X[1]\*\*2-1)-J\_inv[1,1]\*(X[1]-np.tan(X[0]))

def phi(X):

    return np.array([phi\_1(X), phi\_2(X)])

def abs\_dphi\_1x\_1(x\_1):

    return abs(1-J\_inv[0, 0]\*(4\*x\_1-1)-J\_inv[0, 1]\*(-1/(np.cos(x\_1)\*\*2)))

def abs\_dphi\_1x\_2(x\_2):

    return abs(-J\_inv[0, 0]\*(2\*x\_2)-J\_inv[0, 1])

def abs\_dphi\_2x\_1(x\_1):

    return abs(-J\_inv[1, 0]\*(4\*x\_1-1)-J\_inv[1, 1]\*(-1/(np.cos(x\_1)\*\*2)))

def abs\_dphi\_2x\_2(x\_2):

    return abs(1-J\_inv[1, 0]\*(2\*x\_2)-J\_inv[1, 1])

def Jphi(X):

    return np.array([[abs\_dphi\_1x\_1(X[0]), abs\_dphi\_1x\_2(X[1])],

                     [abs\_dphi\_2x\_1(X[0]), abs\_dphi\_2x\_2(X[1])]])

def normJphi(X):

    return max(Jphi(X)[0, 0]+Jphi(X)[0, 1], Jphi(X)[1, 0]+Jphi(X)[1, 1])

def Simple\_iteration\_method(phi, x0, q, e):

    counter = 0

    x = x0

    print('Итерация '+str(counter)+': x = '+str(x))

    counter += 1

    x\_next = phi(x)

    print('Итерация '+str(counter)+': x = '+str(x\_next))

    counter += 1

    while q/(1-q)\*np.linalg.norm(x\_next-x) > e:

        x = x\_next

        x\_next = phi(x)

        print('Итерация '+str(counter)+': x = '+str(x\_next))

        counter += 1

    return x\_next

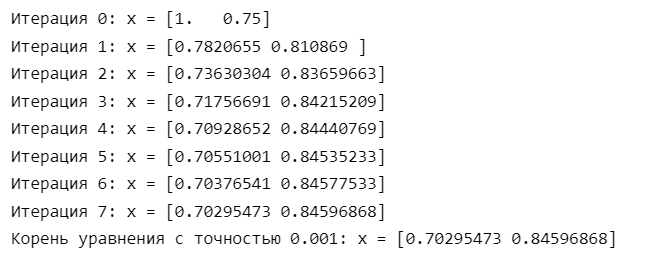
x0 = np.array([1, 0.75])

e = 0.001

answer = Simple\_iteration\_method(phi, x0, q, e)

print('Корень уравнения с точностью '+str(e)+': x = '+str(answer))

Результат работы



**3 Методы приближения функций. Численное дифференцирование и интегрирование.**

**3.1** **Интерполяция**

**Задача**

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

**Вариант 23**

, a)  ; б) ;.

**Интерполяционный многочлен Лагранжа**

Исходный код

class Lagrangian\_interpolation\_polynomial:

    def \_\_init\_\_(self, f, X) -> None:

        self.n = len(X)

        self.f = f

        self.X = X

    def \_\_omega(self, x, i):

        result = 1

        for j, x\_i in enumerate(self.X):

            if i != j:

                result \*= (x-x\_i)

        return result

    def interpolate(self, X):

        result = 0

        for i in range(self.n):

            result += self.f[i]\*self.\_\_omega(X, i)/self.\_\_omega(self.X[i], i)

        return result

def y(x):

    return(1/x)

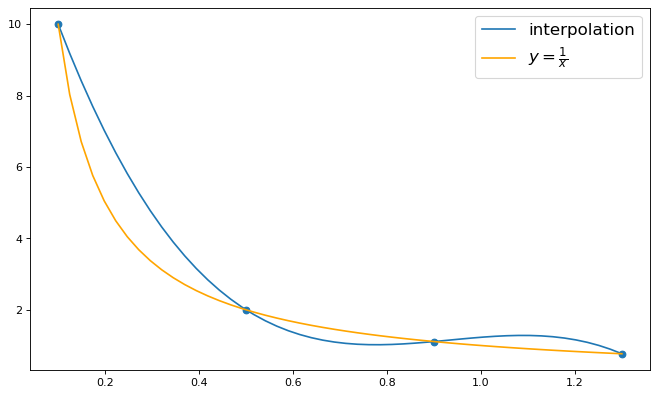
X = np.array([0.1, 0.5, 0.9, 1.3])

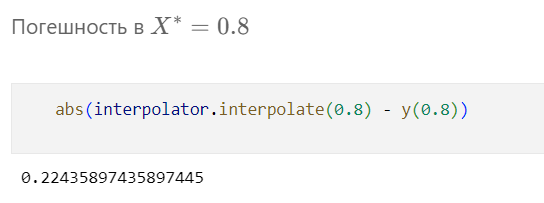
f = y(X)

interpolator = Lagrangian\_interpolation\_polynomial(f, X)

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

Результат работы





**Интерполяционный многочлен Ньютона**

Исходный код

class Newton\_interpolation\_polynomial:

    def \_\_init\_\_(self, f, X) -> None:

        self.n = len(X)

        self.f = f

        self.X = X

        self.divided\_difference = []

        self.divided\_difference.append([self.f[i] for i in range(self.n)])

        for k in range(1, self.n):

            self.divided\_difference.append([])

            for i in range(self.n-k):

                self.divided\_difference[k].append((self.divided\_difference[k-1][i]-self.divided\_difference[k-1][i+1])/ \

                                                  (self.X[i]-self.X[i+k]))

    def interpolate(self, X):

        result = self.f[0]

        for k in range(1, self.n):

            # print(np.array([X-self.X[i] for i in range(self.n)]))

            # print(np.prod(np.array([X-self.X[i] for i in range(self.n)]), axis = 0))

            # print(self.divided\_difference[k, 0])

            result += np.prod(np.array([X-self.X[i] for i in range(k)]), axis = 0)\*self.divided\_difference[k][0]

        return result

def y(x):

    return(1/x)

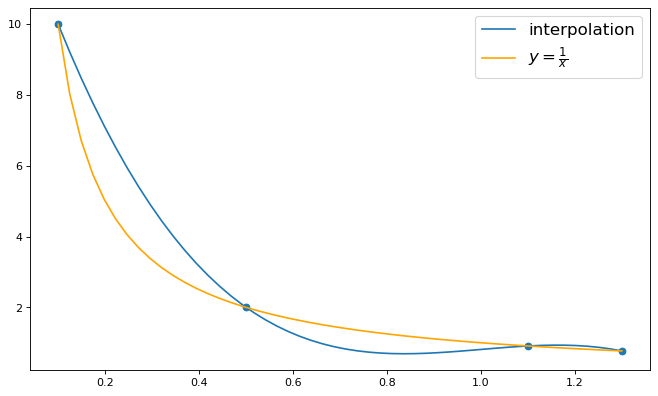
X = np.array([0.1, 0.5, 1.1, 1.3])

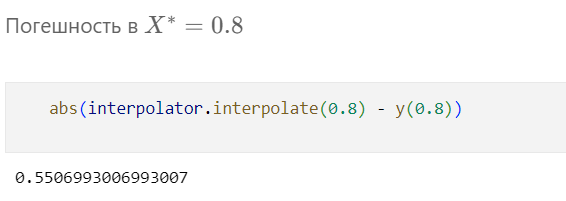
f = y(X)

interpolator = Newton\_interpolation\_polynomial(f, X)

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

Результат работы





**3.2 Сплайн интерполяция**

**Задача**

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

**Вариант 23**

0.8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 0.1 | 0.5 | 0.9 | 1.3 | 1.7 |
|  | 10.0 | 2.0 | 1.1111 | 0.76923 | 0.58824 |

Исходный код

class Cubic\_interpolation\_spline:

    def \_\_init\_\_(self, f, X) -> None:

        self.n = len(X)

        self.f = f

        self.X = X

        A = np.zeros((self.n-1, self.n-1))

        A[0, 0] = 0

        A[1, 1] = 2\*(self.\_\_h(1)+self.\_\_h(2))

        A[1, 2] = self.\_\_h(2)

        for i in range(2, self.n-2):

            A[i, i-1] = self.\_\_h(i-1)

            A[i, i] = 2\*(self.\_\_h(i-1)+self.\_\_h(i))

            A[i, i+1] = self.\_\_h(i)

        A[-1, -2] = self.\_\_h(-2)

        A[-1, -1] = 2\*(self.\_\_h(-2)+self.\_\_h(-1))

        b = np.empty(self.n-1)

        b[0] = 0

        b[1] = 3\*(((self.f[2]-self.f[1])/self.\_\_h(2))-((self.f[1]-self.f[0])/self.\_\_h(1)))

        for i in range(2, self.n-1):

            b[i] = 3\*(((self.f[i+1]-self.f[i])/self.\_\_h(i+1))-((self.f[i]-self.f[i-1])/self.\_\_h(i)))

        b[-1] = 3\*(((self.f[-1]-self.f[-2])/self.\_\_h(-1))-((self.f[-2]-self.f[-3]/self.\_\_h(-2))))

        self.c = np.empty(self.n-1)

        self.c[0] = 0

        self.c[1:] = np.linalg.solve(A[1:, 1:], b[1:])

        self.a = np.array([None]\*(self.n-1))

        self.b = np.array([None]\*(self.n-1))

        self.d = np.array([None]\*(self.n-1))

        for i in range(self.n-2):

            self.a[i] = self.f[i]

            self.b[i] = ((self.f[i+1]-self.f[i])/self.\_\_h(i+1)-(1/3)\*self.\_\_h(i+1)\*(self.c[i+1]+2\*self.c[i]))

            self.d[i] = (self.c[i+1]-self.c[i])/(3\*self.\_\_h(i+1))

        self.a[-1] = self.f[-2]

        self.b[-1] = ((self.f[-1]-self.f[-2])/self.\_\_h(self.n-1))-(2/3)\*self.\_\_h(self.n-1)\*self.c[-1]

        self.d[-1] = -self.c[-1]/(3\*self.\_\_h(self.n-1))

    def \_\_h(self, i):

        return self.X[i]-self.X[i-1]

    def \_\_interpolate\_at\_point(self, x):

        if x < self.X[0] or self.X[-1] < x:

            return None

        for i in range(len(self.X)-1):

            if self.X[i] <= x and x <= self.X[i+1]:

                return self.a[i]+self.b[i]\*(x-self.X[i])+self.c[i]\*(x-self.X[i])\*\*2+self.d[i]\*(x-self.X[i])\*\*3

    def interpolate(self, x):

        if type(x) is float:

            return self.\_\_interpolate\_at\_point(x)

        else:

            result = len(x)\*[None]

            for i in range(len(x)):

                result[i] = self.\_\_interpolate\_at\_point(x[i])

            return np.array(result)

X = np.array([0.1, 0.5, 0.9, 1.3, 1.7])

f = np.array([10, 2, 1,1111, 0.76923, 0.58824])

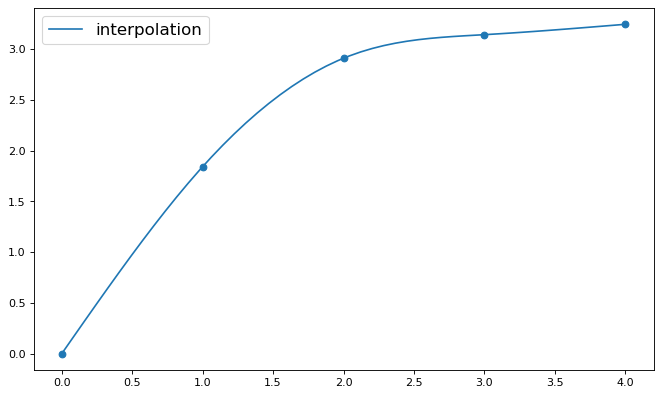
X = np.array([0, 1, 2, 3, 4])

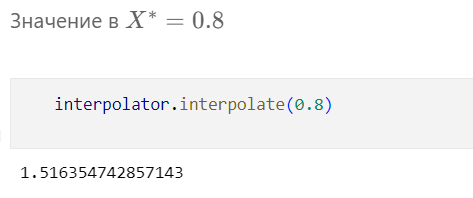
f = np.array([0.0, 1.8415, 2.9093, 3.1411, 3.2432])

interpolator = Cubic\_interpolation\_spline(f, X)

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

Результат работы





**3.3 Метод наименьших квадратов**

**Задача**

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

**Вариант 23**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | 0.1 | 0.5 | 0.9 | 1.3 | 1.7 | 2.1 |
|  | 10. | 2.0 | 1.1111 | 0.76923 | 0.58824 | 0.47619 |

Исходный код

class Linear\_least\_squares:

    def \_\_init\_\_(self, x, y):

        n = len(x)

        A = np.array([[n+1, sum(x)],

                      [sum(x), sum(x\*\*2)]])

        b = np.array([sum(y), sum(y\*x)])

        self.a = np.linalg.solve(A, b)

    def approximate(self, x):

        return self.a[0]+self.a[1]\*x

class Parabolic\_least\_squares:

    def \_\_init\_\_(self, x, y):

        n = len(x)

        A = np.array([[n+1, sum(x), sum(x\*\*2)],

                      [sum(x), sum(x\*\*2), sum(x\*\*3)],

                      [sum(x\*\*2), sum(x\*\*3), sum(x\*\*4)]])

        b = np.array([sum(y), sum(y\*x), sum(y\*x\*\*2)])

        self.a = np.linalg.solve(A, b)

    def approximate(self, x):

        return self.a[0]+self.a[1]\*x+self.a[2]\*x\*\*2

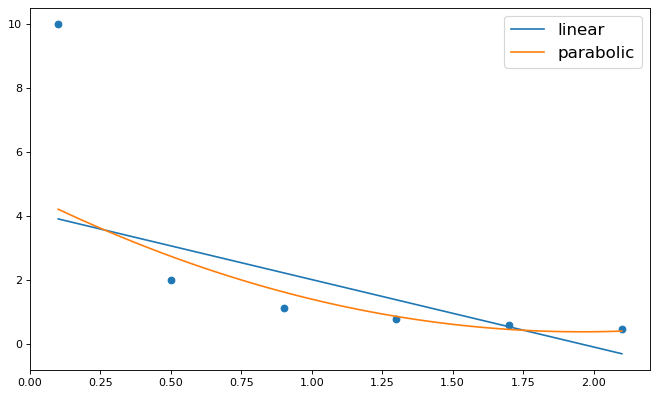
x = np.array([0.1, 0.5, 0.9, 1.3, 1.7, 2.1])

y = np.array([10, 2.0, 1.1111, 0.76923, 0.58824, 0.47619])

linear = Linear\_least\_squares(x, y)

parabolic = Parabolic\_least\_squares(x, y)

Результат работы



**3.4 Численное дифференцирование**

**Задача**

Вычислить первую и вторую производную от таблично заданной функции в точке .

**Вариант 23**

 2.0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 |
|  | 2.0 | 2.1667 | 2.5 | 2.9 | 3.3333 |

Исходный код

def first\_derivative\_first\_order\_left(X, Y, x):

    for i in range(len(X)):

        if x >= X[i] and x < X[i+1]:

            return (Y[i]-Y[i-1])/(X[i]-X[i-1])

def first\_derivative\_first\_order\_right(X, Y, x):

    for i in range(len(X)):

        if x >= X[i] and x < X[i+1]:

            return (Y[i+1]-Y[i])/(X[i+1]-X[i])

def first\_derivative\_second\_order(X, Y, x):

    for i in range(len(X)):

        print()

        if x >= X[i] and x < X[i+1]:

            return (Y[i+1]-Y[i])/(X[i+1]-X[i])+(((Y[i+2]-Y[i+1])/(X[i+2]-X[i+1])-(Y[i+1]-Y[i])/(X[i+1]-X[i]))/(X[i+2]-X[i]))\*(2\*x-X[i]-X[i+1])

def second\_derivative\_second\_order(X, Y, x):

    for i in range(len(X)):

        if x >= X[i] and x < X[i+1]:

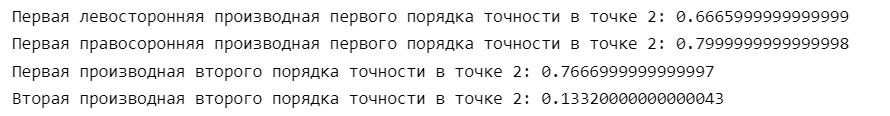
            return 2\*((Y[i+2]-Y[i+1])/(X[i+2]-X[i+1])-(Y[i+1]-Y[i])/(X[i+1]-X[i]))/(X[i+2]-X[i])

X = np.array([1.0, 1.5, 2.0, 2.5, 3.0])

Y = np.array([2.0, 2.1667, 2.5, 2.9, 3.3333])

x = 2

Результат работы



**3.5 Численное интегрирование**

**Задача**

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

**Вариант 23**

, 

Исходный код

def rectangle\_integration(f, x\_0, x\_k, h):

    x = np.arange(x\_0, x\_k+h, h)

    return h\*sum([f((x[i]+x[i+1])/2) for i in range(len(x)-1)])

def trapezoidal\_integration(f, x\_0, x\_k, h):

    x = np.arange(x\_0, x\_k+h, h)

    F = h\*(f(x\_0)/2+f(x\_k)/2)

    return F+h\*sum([f(x[i]) for i in range(1, len(x)-1)])

def simpson\_integration(f, x\_0, x\_k, h):

    x = np.arange(x\_0, x\_k+h, h)

    F = (h/3)\*(f(x\_0)+f(x\_k))

    return (h/3)\*(sum([4\*f(x[i])+2\*f(x[i+1]) for i in range(1, len(x)-2, 2)])+4\*f(x[-2]))+F

def runge\_romberg\_richardson\_refinement(F\_h, F\_kh, k, p):

    return F\_h+(F\_h-F\_kh)/(k\*\*p-1)

def y(x):

    return x/(3\*x+4)\*\*2

x\_0 = -1

x\_k = 1

h = 0.5

rectangle\_result = rectangle\_integration(y, x\_0, x\_k, h)

trapezoidal\_result = trapezoidal\_integration(y, x\_0, x\_k, h)

simpson\_result = simpson\_integration(y, x\_0, x\_k, h)

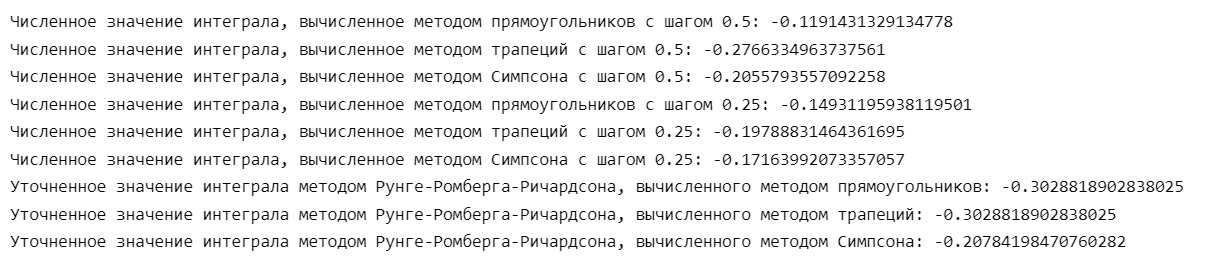
h = 0.25

rectangle\_result\_double\_precision = rectangle\_integration(y, x\_0, x\_k, h)

trapezoidal\_result\_double\_precision = trapezoidal\_integration(y, x\_0, x\_k, h)

simpson\_result\_double\_precision = simpson\_integration(y, x\_0, x\_k, h)

Результат работы



### 4 Численные методы решения начальных и краевых задач для ОДУ.

**4.1 Численные методы решения задачи Коши**

**Задача**

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

**Вариант 23**

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

**Метод Эйлера**

Исходный код

def euler\_method(f, y\_0, dy\_0, h, left\_x, right\_x):

    x = np.arange(left\_x, right\_x+h, h)

    y = np.empty\_like(x)

    z = np.empty\_like(x)

    y[0] = y\_0

    z[0] = dy\_0

    for i in range(1, len(x)):

        z[i] = z[i-1]+h\*f(x[i-1], y[i-1], z[i-1])

        y[i] = y[i-1]+h\*z[i-1]

    return x, y

def true\_y(x):

    return x\*\*2+x+(1/x)

def f(x, y, z):

    return (-x\*z+y+3\*x\*\*2)/(x\*\*2)

y\_0 = 3

dy\_0 = 2

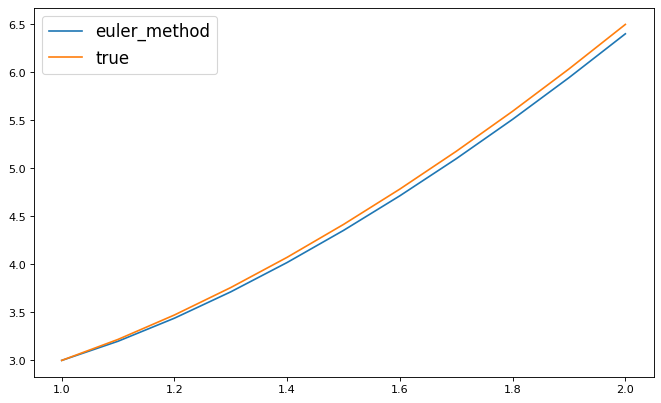
left\_x = 1

right\_x = 2

h = 0.1

x, y = euler\_method(f, y\_0, dy\_0, h, left\_x, right\_x)

Результат работы



**Метод Метод Рунге-Ромберга**

Исходный код

def runge\_kutti\_method(f, y\_0, dy\_0, h, left\_x, right\_x):

    x = np.arange(left\_x, right\_x+h, h)

    y = np.empty\_like(x)

    z = np.empty\_like(x)

    y[0] = y\_0

    z[0] = dy\_0

    for i in range(1, len(x)):

        K1 = h\*z[i-1]

        L1 = h\*f(x[i-1], y[i-1], z[i-1])

        K2 = h\*(z[i-1]+(1/2)\*L1)

        L2 = h\*f(x[i-1]+(1/2)\*h, y[i-1]\*(1/2)\*K1, z[i-1]+(1/2)\*L1)

        K3 = h\*(z[i-1]+(1/2)\*L2)

        L3 = h\*f(x[i-1]+(1/2)\*h, y[i-1]+(1/2)\*K2, z[i-1]+(1/2)\*L2)

        K4 = h\*(z[i-1]+L3)

        L4 = h\*f(x[i-1]+h, y[i-1]+K3, z[i-1]+L3)

        delta\_y = (1/6)\*(K1+2\*K2+2\*K3+K4)

        delta\_z = (1/6)\*(L1+2\*L2+2\*L3+L4)

        y[i] = y[i-1]+delta\_y

        z[i] = z[i-1]+delta\_z

    return x, y

def f(x, y, z):

    return (2\*x\*z)/(x\*\*2+1)

y\_0 = 3

dy\_0 = 2

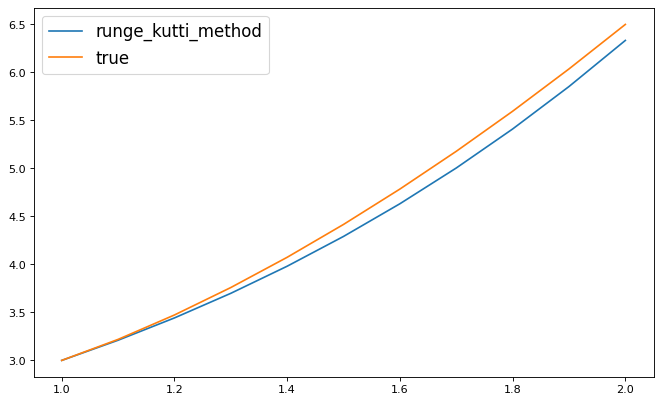
left\_x = 1

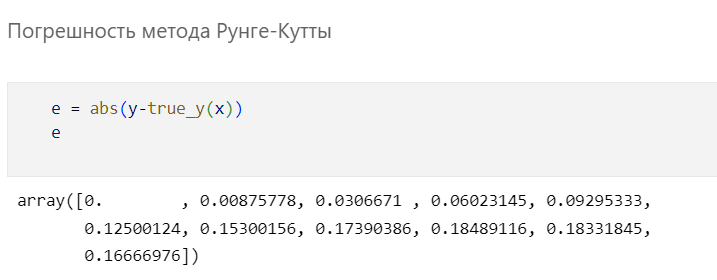
right\_x = 2

h = 0.1

x, y = runge\_kutti\_method(f, y\_0, dy\_0, h, left\_x, right\_x)

Результат работы





**Метод Адамса**

Исходный код

def adams\_method(f, y\_0, dy\_0, h, left\_x, right\_x):

    def runge\_kutti\_method(f, y\_0, dy\_0, h, left\_x, right\_x):

        x = np.arange(left\_x, right\_x+h, h)

        y = np.empty\_like(x)

        z = np.empty\_like(x)

        y[0] = y\_0

        z[0] = dy\_0

        for i in range(1, len(x)):

            K1 = h\*z[i-1]

            L1 = h\*f(x[i-1], y[i-1], z[i-1])

            K2 = h\*(z[i-1]+(1/2)\*L1)

            L2 = h\*f(x[i-1]+(1/2)\*h, y[i-1]\*(1/2)\*K1, z[i-1]+(1/2)\*L1)

            K3 = h\*(z[i-1]+(1/2)\*L2)

            L3 = h\*f(x[i-1]+(1/2)\*h, y[i-1]+(1/2)\*K2, z[i-1]+(1/2)\*L2)

            K4 = h\*(z[i-1]+L3)

            L4 = h\*f(x[i-1]+h, y[i-1]+K3, z[i-1]+L3)

            delta\_y = (1/6)\*(K1+2\*K2+2\*K3+K4)

            delta\_z = (1/6)\*(L1+2\*L2+2\*L3+L4)

            y[i] = y[i-1]+delta\_y

            z[i] = z[i-1]+delta\_z

        return y, z

    start\_y, start\_z = runge\_kutti\_method(f, y\_0, dy\_0, h, left\_x, left\_x+2\*h)

    x = np.arange(left\_x, right\_x+h, h)

    y = np.empty\_like(x)

    z = np.empty\_like(x)

    y[0] = y\_0

    z[0] = dy\_0

    for i in range(1, 4):

        y[i] = start\_y[i]

        z[i] = start\_z[i]

    for i in range(4, len(x)):

        z[i] = z[i-1]+(h/24)\*(55\*f(x[i-1], y[i-1], z[i-1])-59\*f(x[i-2], y[i-2], z[i-2])+37\*f(x[i-3], y[i-3], z[i-3])-9\*f(x[i-4], y[i-4], z[i-4]) )

        y[i] = y[i-1]+h\*z[i-1]

    return x, y

def f(x, y, z):

    return (2\*x\*z)/(x\*\*2+1)

y\_0 = 3

dy\_0 = 2

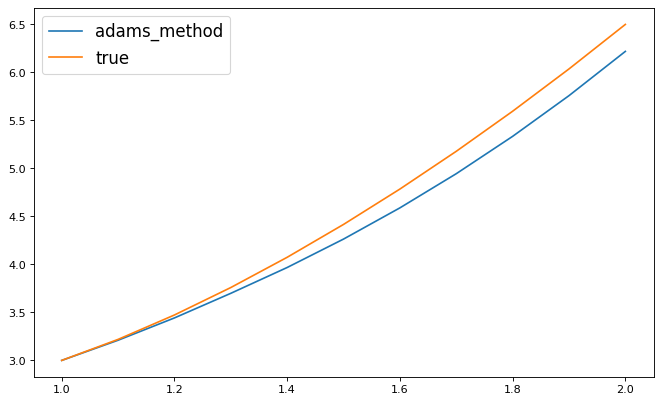
left\_x = 1

right\_x = 2

h = 0.1

x, y = adams\_method(f, y\_0, dy\_0, h, left\_x, right\_x)

Результат работы



**4.2 Численные методы решения краевой задачи для ОДУ**

**Задача**

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

**Вариант 12**

|  |  |
| --- | --- |
| x(x-1)y″-xy′+ y=0  y′(1)=3  y(3)-3y′(3)=-4 | y(x)=2+x+2x ln|x| |

**Метод стрельбы**

Исходный код

def shooting\_method(f, y\_0, h, left\_x, right\_x, end\_value, e):

    def runge\_kutti\_method(f, y\_0, dy\_0, h, left\_x, right\_x):

        x = np.arange(left\_x, right\_x+h, h)

        y = np.empty\_like(x)

        z = np.empty\_like(x)

        y[0] = y\_0

        z[0] = dy\_0

        for i in range(1, len(x)):

            K1 = h\*z[i-1]

            L1 = h\*f(x[i-1], y[i-1], z[i-1])

            K2 = h\*(z[i-1]+(1/2)\*L1)

            L2 = h\*f(x[i-1]+(1/2)\*h, y[i-1]\*(1/2)\*K1, z[i-1]+(1/2)\*L1)

            K3 = h\*(z[i-1]+(1/2)\*L2)

            L3 = h\*f(x[i-1]+(1/2)\*h, y[i-1]+(1/2)\*K2, z[i-1]+(1/2)\*L2)

            K4 = h\*(z[i-1]+L3)

            L4 = h\*f(x[i-1]+h, y[i-1]+K3, z[i-1]+L3)

            delta\_y = (1/6)\*(K1+2\*K2+2\*K3+K4)

            delta\_z = (1/6)\*(L1+2\*L2+2\*L3+L4)

            y[i] = y[i-1]+delta\_y

            z[i] = z[i-1]+delta\_z

        return y

    n0, n = 1, 0.8

    solution0 = runge\_kutti\_method(f, y\_0, n0, h, left\_x, right\_x)

    solution0\_y = solution0[-1]

    solution = runge\_kutti\_method(f, y\_0, n, h, left\_x, right\_x)

    solution\_y = solution[-1]

    prev\_n = n0

    prev\_solution\_y = solution0\_y

    while abs(abs(solution\_y-end\_value) >= e):

        next\_n = n-((n-prev\_n)/(solution\_y-prev\_solution\_y))\*(solution\_y-end\_value)

        prev\_n = n

        n = next\_n

        prev\_solution = runge\_kutti\_method(f, y\_0, prev\_n, h, left\_x, right\_x)

        prev\_solution\_y = prev\_solution[-1]

        solution = runge\_kutti\_method(f, y\_0, n, h, left\_x, right\_x)

        solution\_y = solution[-1]

    return np.arange(left\_x, right\_x+h, h), solution

def true\_y(x):

    return np.exp(-x)/x

def f(x, y, z):

    return -2\*(z/x) + y

y\_0 = 1/np.e

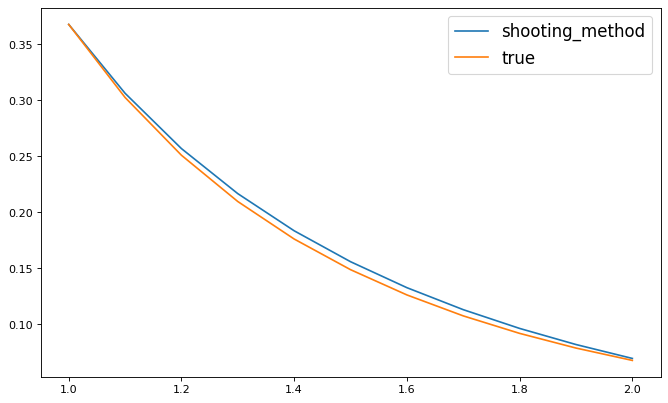
end\_value = 0.5\*(1/np.e\*\*2)

left\_x = 1

right\_x = 2

x, y = shooting\_method(f, y\_0, h, left\_x, right\_x, end\_value, e)

Результат работы



Метод конечных разностей

def finite\_difference\_method(p, q, h, left\_x, right\_x, alpha\_left, beta\_left, y\_0, alpha\_right, beta\_right, c):

    x = np.arange(left\_x, right\_x+h, h)

    n = len(x)

    A = np.zeros((n, n))

    A[0, 0] = ((-3\*alpha\_left)/(2\*h))+beta\_left

    A[0, 1] = (4\*alpha\_left)/(2\*h)

    A[0, 2] = -(alpha\_left)/(2\*h)

    A[-1, -3] = alpha\_right/(2\*h)

    A[-1, -2] = (-4\*alpha\_right)/(2\*h)

    A[-1, -1] = ((3\*alpha\_right)/(2\*h))+beta\_right

    for i in range(1, n-1):

        A[i, i-1] = ((1/(h\*\*2))-p(x[i])/(2\*h))

        A[i, i] = ((-2/(h\*\*2))+q(x[i]))

        A[i, i+1] = ((1/(h\*\*2))+p(x[i])/(2\*h))

    b = np.zeros(n)

    b[0] = y\_0

    b[-1] = c

    y = np.linalg.solve(A, b)

    return x, y

def true\_y(x):

    return 2+x+2\*x\*np.log(abs(x))

def p(x):

    return -1/(x-1)

def q(x):

    return 1/(x\*(x-1))

y\_0 = 3

c = -4

h = 0.1

left\_x = 1

right\_x = 3

alpha\_left = 0

beta\_left = 1

alpha\_right = -3

beta\_right = 1

x, y = finite\_difference\_method(p, q, h, left\_x, right\_x, alpha\_left, beta\_left, y\_0, alpha\_right, beta\_right, c)

Результат работы

