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

1.1 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

$$\begin{cases} 2 \cdot x_1 - 7 \cdot x_2 + 8 \cdot x_3 - 4 \cdot x_4 = 57 \\ -x_2 + 4 \cdot x_3 - x_4 = 24 \\ 3 \cdot x_1 - 4 \cdot x_2 + 2 \cdot x_3 - x_4 = 28 \\ -9 \cdot x_1 + x_2 - 4 \cdot x_3 + 6 \cdot x_4 = 12 \end{cases}$$

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.

1.2 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

$$\begin{cases} 7 \cdot x_1 - 5 \cdot x_2 = 38 \\ -6 \cdot x_1 + 19 \cdot x_2 - 9 \cdot x_3 = 14 \\ 6 \cdot x_2 - 18 \cdot x_3 + 7 \cdot x_4 = -45 \\ -7 \cdot x_3 - 11 \cdot x_4 - 2 \cdot x_5 = 30 \\ 5 \cdot x_4 - 7 \cdot x_5 = 48 \end{cases}$$

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 n^3.

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.

1.3 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

$$\begin{cases} -24 \cdot x_1 - 6 \cdot x_2 + 4 \cdot x_3 + 7 \cdot x_4 = 130 \\ -8 \cdot x_1 + 21 \cdot x_2 + 4 \cdot x_3 - 2 \cdot x_4 = 139 \\ 6 \cdot x_1 + 6 \cdot x_2 + 16 \cdot x_3 = -84 \\ -7 \cdot x_1 - 7 \cdot x_2 + 5 \cdot x_3 + 24 \cdot x_4 = -165 \end{cases}$$

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.

1.4 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

$$\begin{pmatrix} 9 & -5 & -6 \\ -5 & 1 & -8 \\ -6 & -8 & -3 \end{pmatrix}$$

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 $U^{-1} = U^T$, то $\Lambda = U^T A U$,

где Λ - диагональная матрица с собственными значениями на главной диагонали

1.5 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

$$\begin{pmatrix} 1 & 5 & -6 \\ 9 & -7 & -9 \\ 6 & -1 & -9 \end{pmatrix}$$

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

$$\ln(x+2) - x^4 + 0.5 = 0$$

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

```
Итерация 0: x = 1.3
Итерация 1: x = 1.1630310333409177
Итерация 2: x = 1.133229753109308
Итерация 3: x = 1.1319338486866162
Итерация 4: x = 1.1319314744236986
Корень уравнения с точностью 0.001: x = 1.1319314744236986
```

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

```
Итерация 0: x = 1.0
Итерация 1: x = 1.1244387062030958
Итерация 2: x = 1.1315183627844028
Корень уравнения с точностью 0.001: x = 1.1315183627844028
```

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

$$\begin{cases} ax_1^2 - x_1 + x_2^2 - 1 = 0, \\ x_2 - \operatorname{tg} x_1 = 0. \end{cases}$$, где $a = 2$.

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

```
Итерация 0: x = [0.7 0.8]
Итерация 1: x = [0.70272068 0.84693924]
Итерация 2: x = [0.70224688 0.84613628]
Корень уравнения с точностью 0.001: x = [0.70224688 0.84613628]
```

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

```
Итерация 0: x = [1.   0.75]
Итерация 1: x = [0.7820655 0.810869 ]
Итерация 2: x = [0.73630304 0.83659663]
Итерация 3: x = [0.71756691 0.84215209]
Итерация 4: x = [0.70928652 0.84440769]
Итерация 5: x = [0.70551001 0.84535233]
Итерация 6: x = [0.70376541 0.84577533]
Итерация 7: x = [0.70295473 0.84596868]
Корень уравнения с точностью 0.001: x = [0.70295473 0.84596868]
```

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

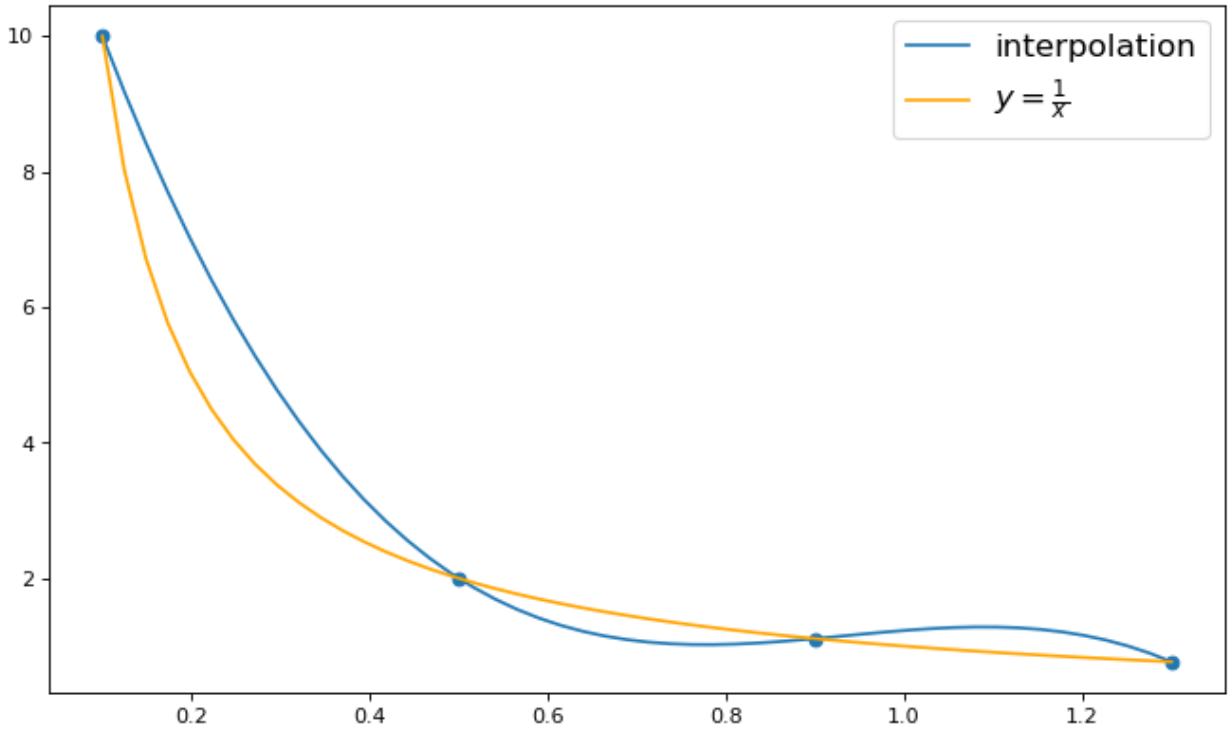
$y = \frac{1}{x}$, а) $X_i = 0.1, 0.5, 0.9, 1.3$; б) $X_i = 0.1, 0.5, 1.1, 1.3; X^* = 0.8$.

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



Погешность в $X^* = 0.8$

```
abs(interpolator.interpolate(0.8) - y(0.8))
```

```
0.22435897435897445
```

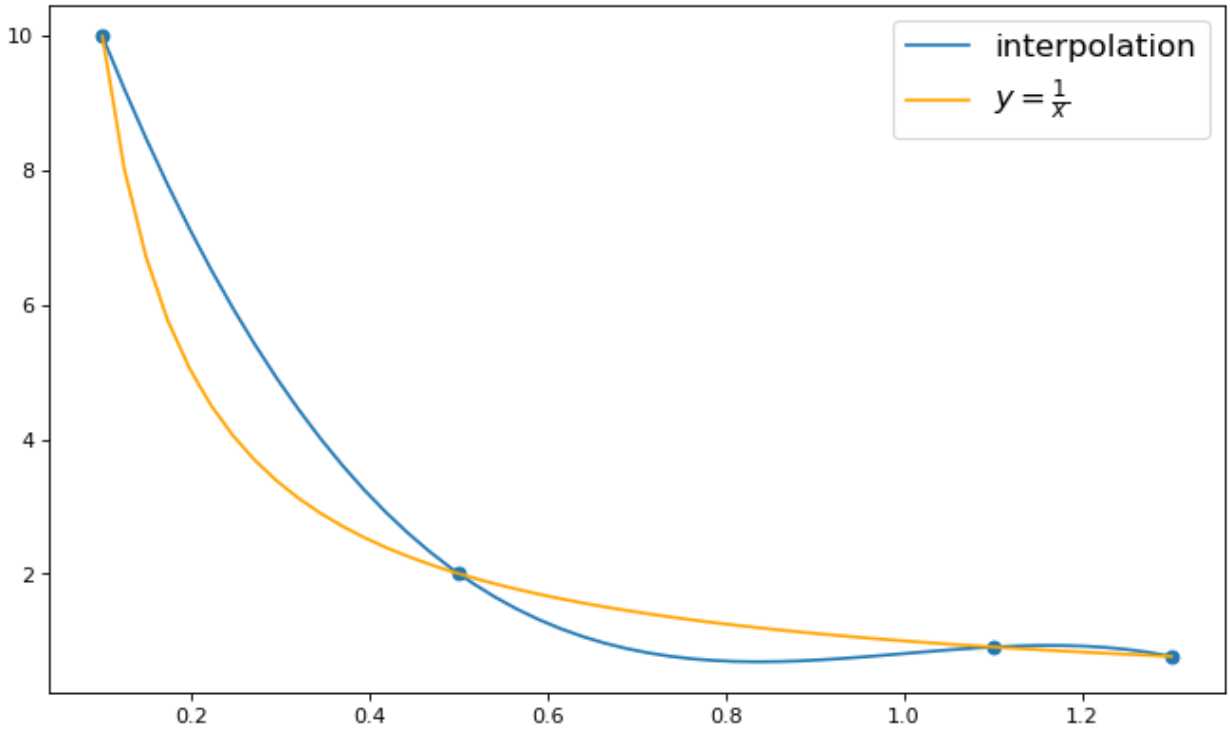
Интерполяционный многочлен Ньютона

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



Погешность в $X^* = 0.8$

```
abs(interpolator.interpolate(0.8) - y(0.8))
```

```
0.5506993006993007
```

3.2 Сплайн интерполяция

Задача

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

Вариант 23

$X^* = 0.8$

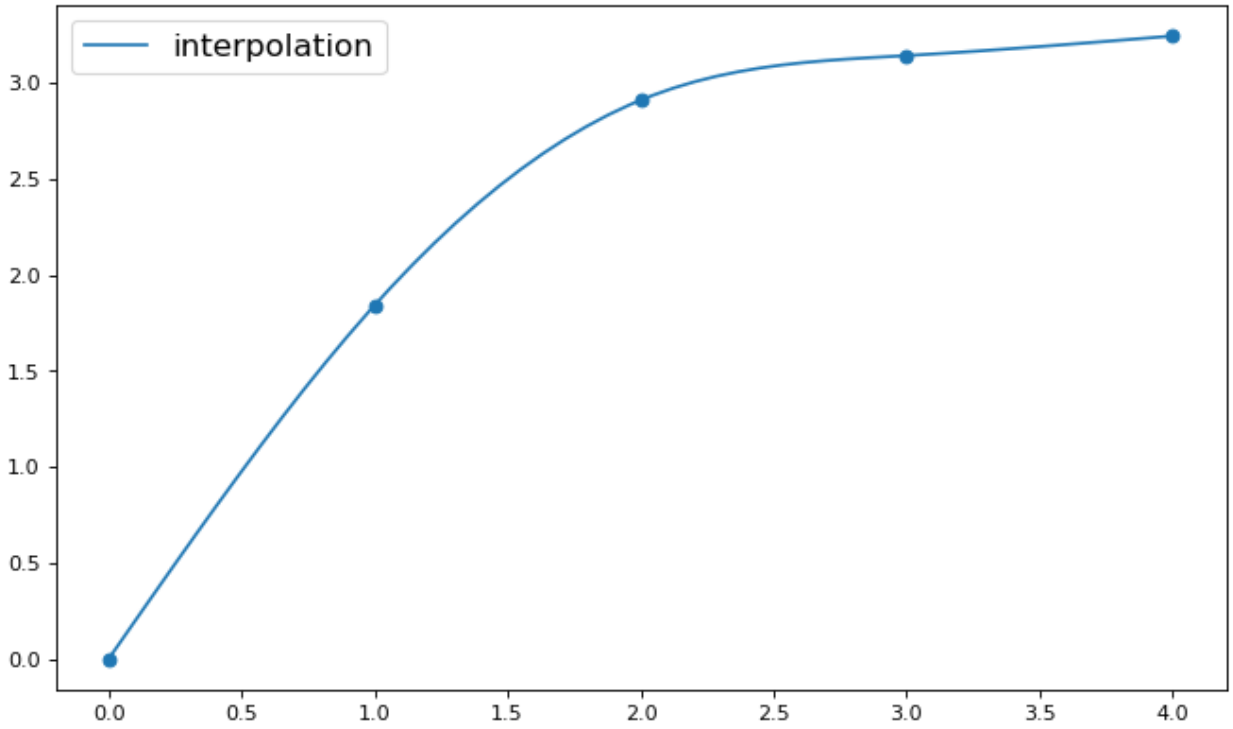
i	0	1	2	3	4
x_i	0.1	0.5	0.9	1.3	1.7
f_i	10.0	2.0	1.1111	0.76923	0.58824

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



Значение в $X^* = 0.8$

```
interpolator.interpolate(0.8)
```

```
1.516354742857143
```

3.3 Метод наименьших квадратов

Задача

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

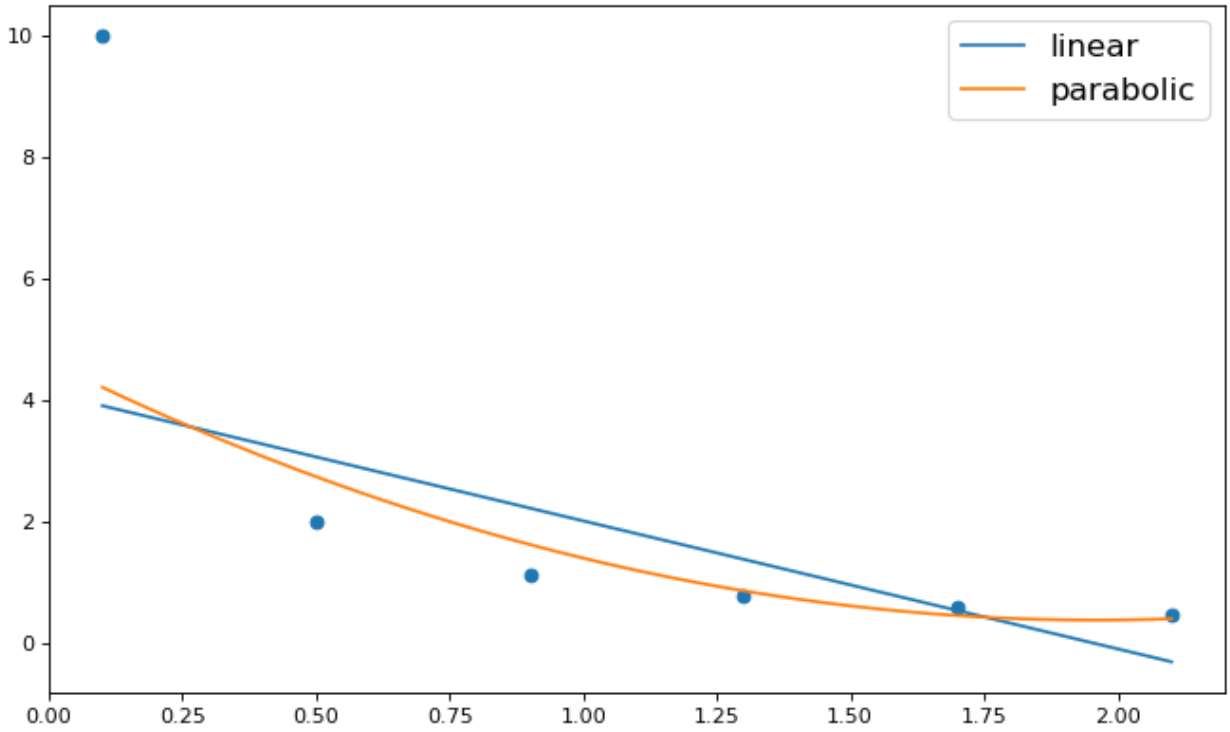
Вариант 23

i	0	1	2	3	4	5
x_i	0.1	0.5	0.9	1.3	1.7	2.1
y_i	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 $y_i = f(x_i)$, $\mathrm{i} = 0,1,2,3,4$ в точке $x = X^*$.

Вариант 23

$X^* = 2.0$

i	0	1	2	3	4
x_i	1.0	1.5	2.0	2.5	3.0
y_i	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

```
Первая левосторонняя производная первого порядка точности в точке 2: 0.6665999999999999
Первая правосоронняя производная первого порядка точности в точке 2: 0.7999999999999998
Первая производная второго порядка точности в точке 2: 0.7666999999999997
Вторая производная второго порядка точности в точке 2: 0.13320000000000043
```

3.5 Численное интегрирование

Задача

Вычислить определенный интеграл $F = \int_{X_0}^{X_1} y\,dx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга.

Вариант 23

$$y = \frac{1}{\sqrt{(2x+7)(3x+4)}}, \qquad X_0 = 0, \quad X_k = 4, \quad h_1 = 1.0, \quad h_2 = 0.5$$

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

```
Численное значение интеграла, вычисленное методом прямоугольников с шагом 0.5: -0.1191431329134778
Численное значение интеграла, вычисленное методом трапеций с шагом 0.5: -0.2766334963737561
Численное значение интеграла, вычисленное методом Симпсона с шагом 0.5: -0.2055793557092258
Численное значение интеграла, вычисленное методом прямоугольников с шагом 0.25: -0.14931195938119501
Численное значение интеграла, вычисленное методом трапеций с шагом 0.25: -0.19788831464361695
Численное значение интеграла, вычисленное методом Симпсона с шагом 0.25: -0.17163992073357057
Уточненное значение интеграла методом Рунге-Ромберга-Ричардсона, вычисленного методом прямоугольников: -0.3028818902838025
Уточненное значение интеграла методом Рунге-Ромберга-Ричардсона, вычисленного методом трапеций: -0.3028818902838025
Уточненное значение интеграла методом Рунге-Ромберга-Ричардсона, вычисленного методом Симпсона: -0.20784198470760282
```

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

$x^2y''+xy'-y-3x^2=0,$ $y(1)=3,$ $y'(1)=2,$ $x\in[1,2], h=0.1$	$y=x^2+x+\frac{1}{x}$

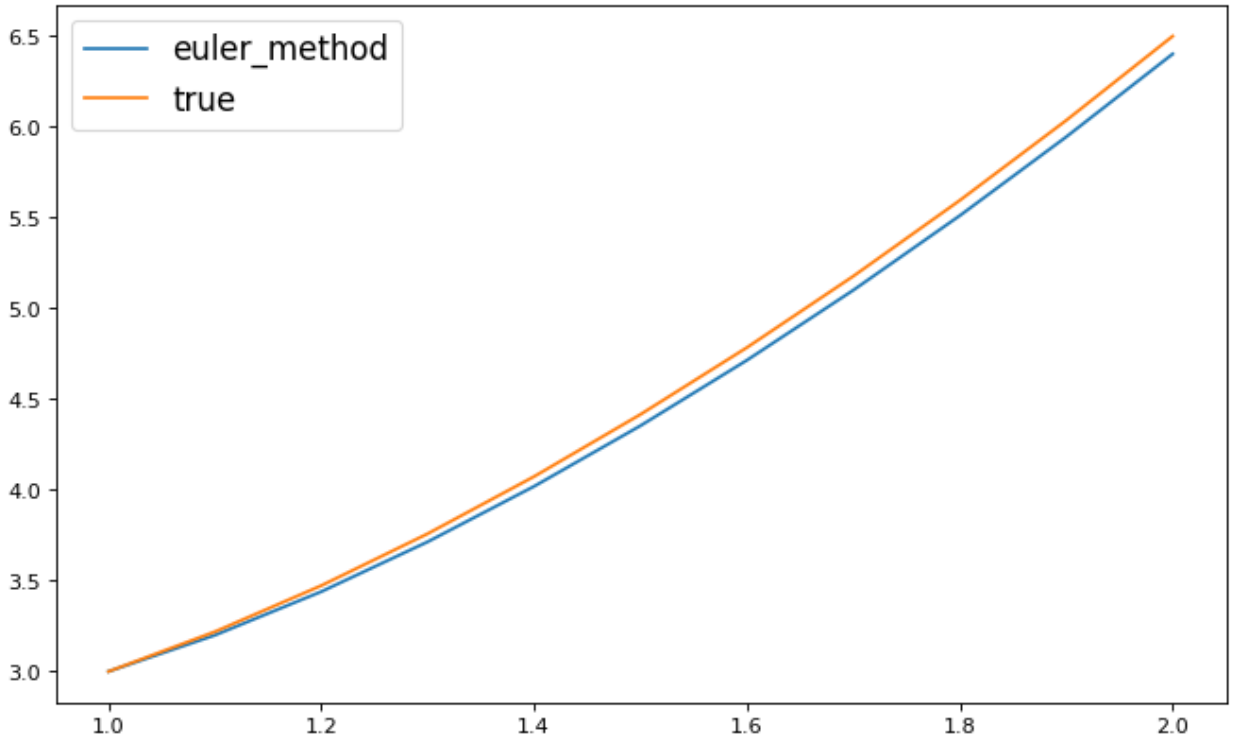
Метод Эйлера

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



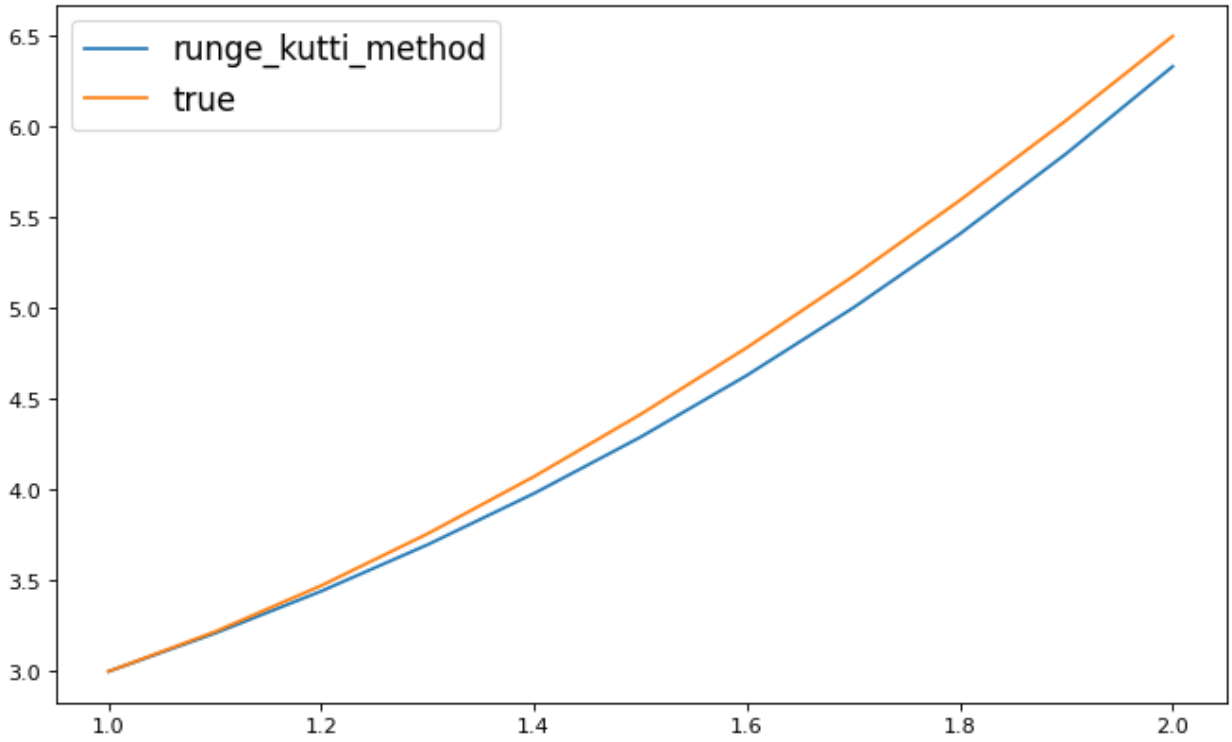
Метод Метод Рунге-Ромберга

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



Погрешность метода Рунге-Кутты

```
e = abs(y-true_y(x))
e
```

```
array([0.        , 0.00875778, 0.0306671 , 0.06023145, 0.09295333,
       0.12500124, 0.15300156, 0.17390386, 0.18489116, 0.18331845,
       0.16666976])
```

Метод Адамса

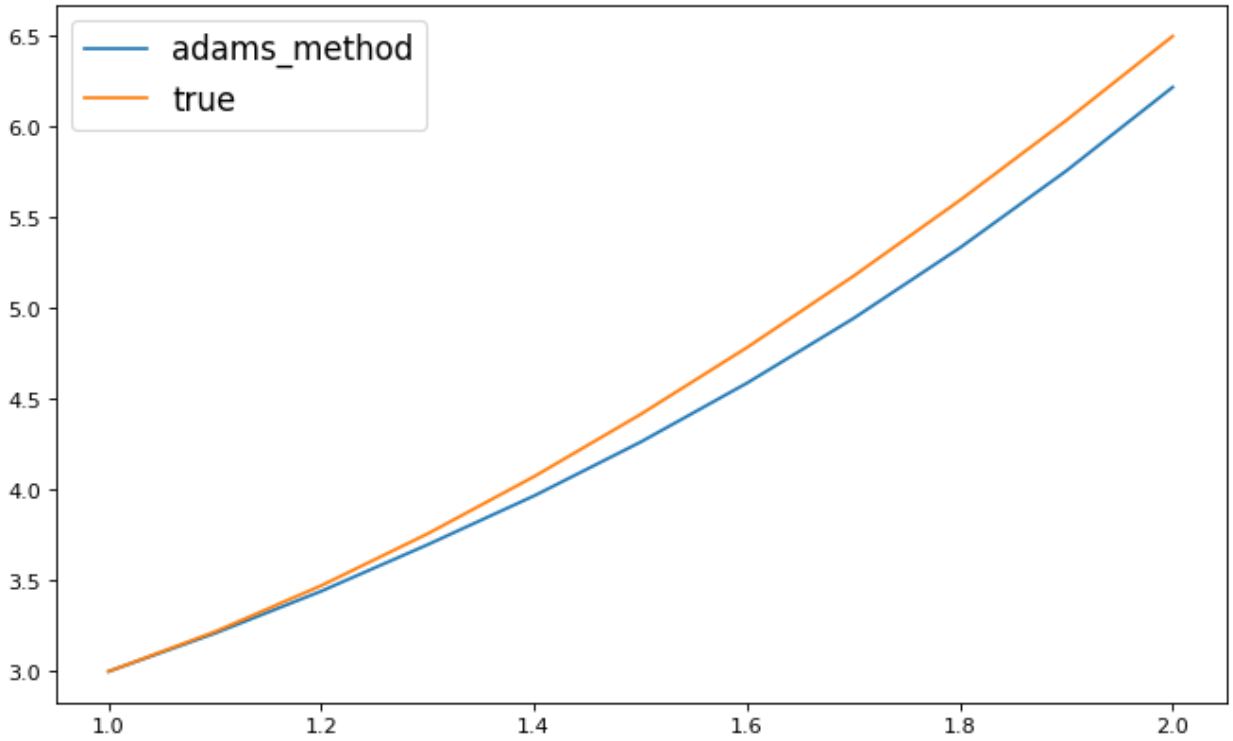
Исходный код

```
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

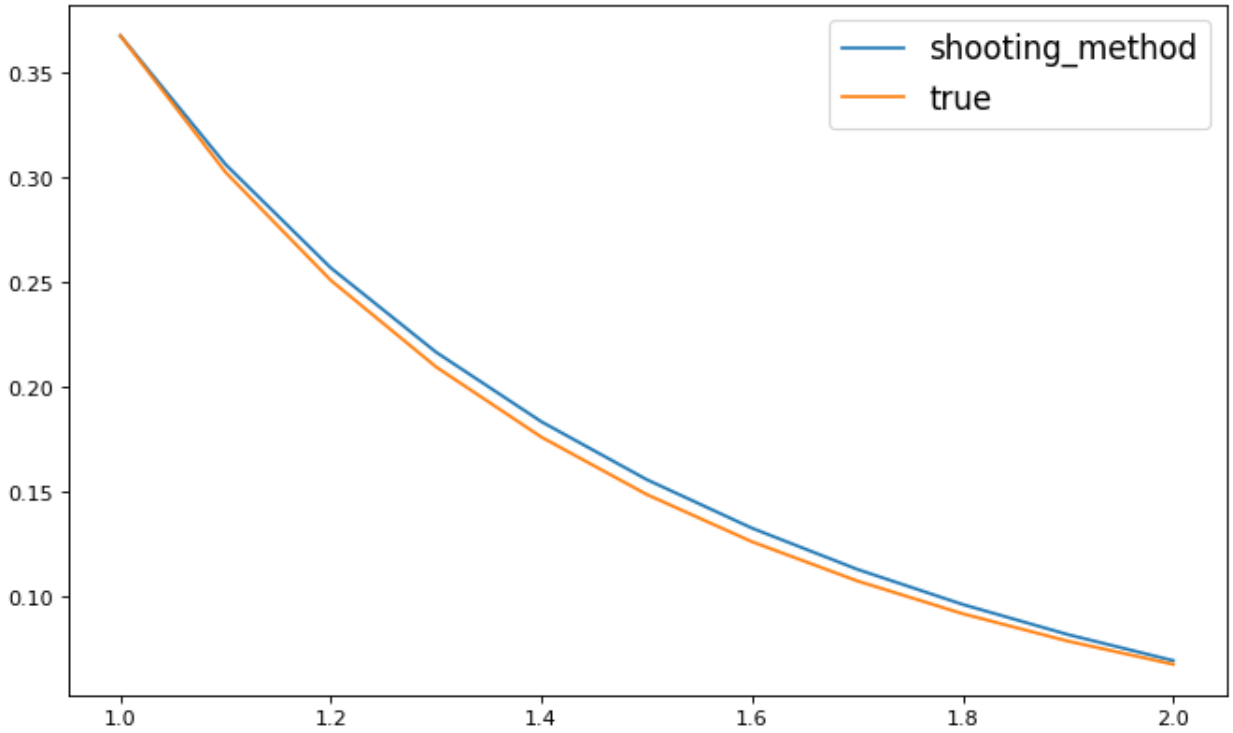
Исходный код

```
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