Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По дисциплине «Методы численного анализа»

По теме «Решение краевых задач методом разностных аппроксимаций»

Вариант 6

Выполнил:

студент гр. 653504

Куликов А.Д.

Проверил:

Анисимов В.Я.

Минск 2018

**Краткие теоретические сведения**

**Постановка задачи:**

Будем рассматривать дифференциальное уравнение второго порядка.



где  − заданные непрерывные на отрезке  *[a, b]* функции.

*Краевой задачей* называется задача нахождения решения , удовлетворяющего граничным условиям:



Если на границах х = а и х = b заданы значения искомой функции у(а), у(b), то такие условия называются граничными условиями первого рода, а задача называется первой краевой задачей для ОДУ.

Граничные условия второго рода:



Граничные условия третьего рода (если на границах заданы линейные комбинации искомой функции и ее первой производной):



Рассмотрим дифференциальное уравнение  первого порядка  


Пусть его решением является функция *U=U(x).*

Разобьем отрезок *[a,b]* на *n* равных частей с шагом *h*, то есть построим последовательность {*xk*}:



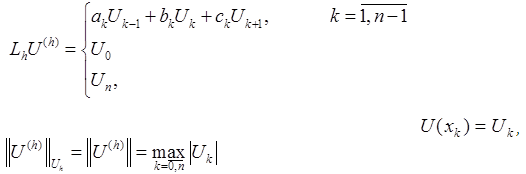
Множество узлов *xk* на отрезке *[a,b]* образует *сетку*  с шагом *h*.

При этом, любую функцию *gk*   *k=0, 1,…,n* , определенную на сетке , будем называть сеточной функцией. В частности, сеточной  функцией будет последовательность  порожденная решением  дифференциаль-ного уравнения.

Заменим производные функции во внутренних узлах их разностными аппроксимациями:



Значение функции*U* в *k*–ом узле обозначим



а функции *f* соответственно через *.*  В новых обозначениях получим разностное уравнение:



C большей точностью первую производную можно также представить в виде:



тогда уравнение будет иметь вид



Рассмотрим теперь на данном отрезке дифференциальное уравнение

второго порядка



Аналогично, заменив производные их разностными аппроксимациями, получим:



Преобразуем полученное разностное уравнение:



Полученные разностные уравнения будем также называть *разностными* *схемами*для соответствующих дифференциальных уравнений. Таким образом,   мы перешли от дифференциальных  уравнений  к разностным уравнениям вида:

                                                              (1)

и

                                              (2)

где   −  некоторые коэффициенты.  Уравнения (1) и (2) будем называть соответственно *линейными разностными уравнениями первого и второго порядка* (при условии, что      в  (1) и      в    (2)). Заметим, что при переходе от дифференциального уравнения к разностному уравнению, порядок уравнения не обязательно сохраняется.

Рассмотрим задачу вида:



называемую разностной краевой задачей (РКЗ). Данную задачу можно коротко переписать в виде:



где линейный разностный оператор  или   имеет вид:



Задачу РКЗ будем называть хорошо обусловленной, если она имеет решение, причем единственное, при любых значениях , и, кроме того, это решение {Uk} удовлетворяет соотношению:



где *M = const* ,   не зависящая от  *n.*

Теорема1 (достаточный признак хорошей обусловленности). Пусть выполнимо следующее условие:



Тогда задача РКЗ хорошо обусловлена, а условие хорошей обусловленности имеет следующий вид:



Теорема 2 (критерий хорошей обусловленности). Пусть . Тогда задача является хорошо обусловленной тогда и только тогда, когда корни *q1*,*q2* характеристического уравнения удовлетворяют условиям  .

Разностная схема:



называется устойчивой, если существует число  такое, что при любом  разностная задача имеет единственное решение при любой правой части  f(h), и это решение удовлетворяет соотношению:



где С – независимая от шага *h* константа.

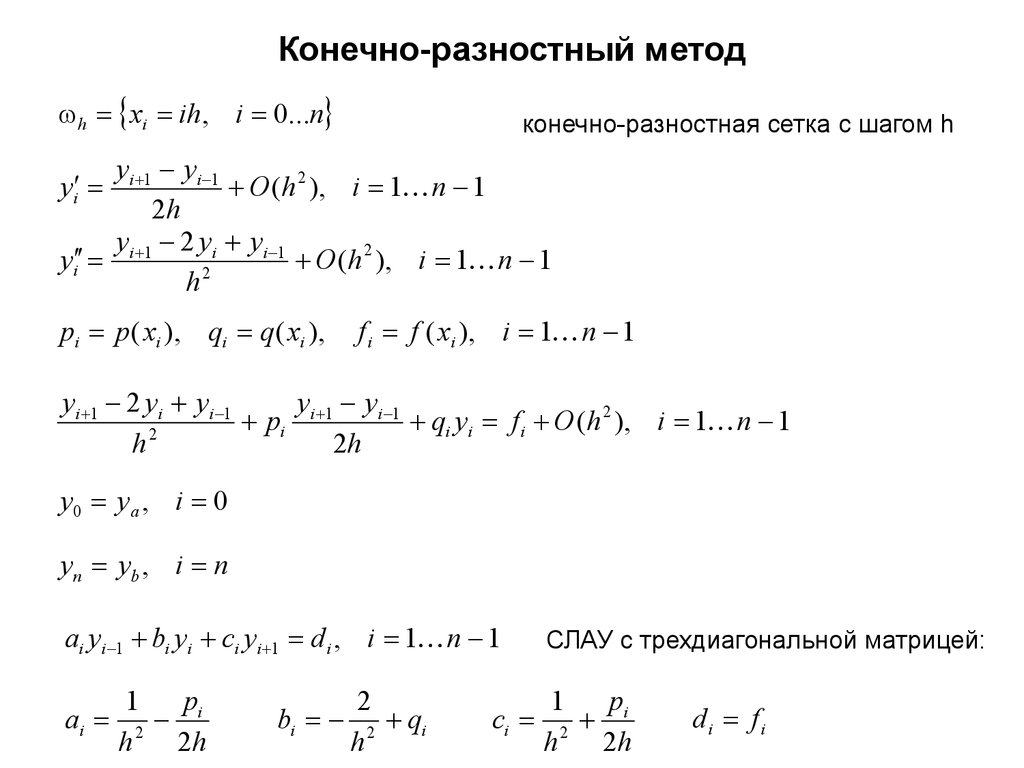
**Метод прогонки**.

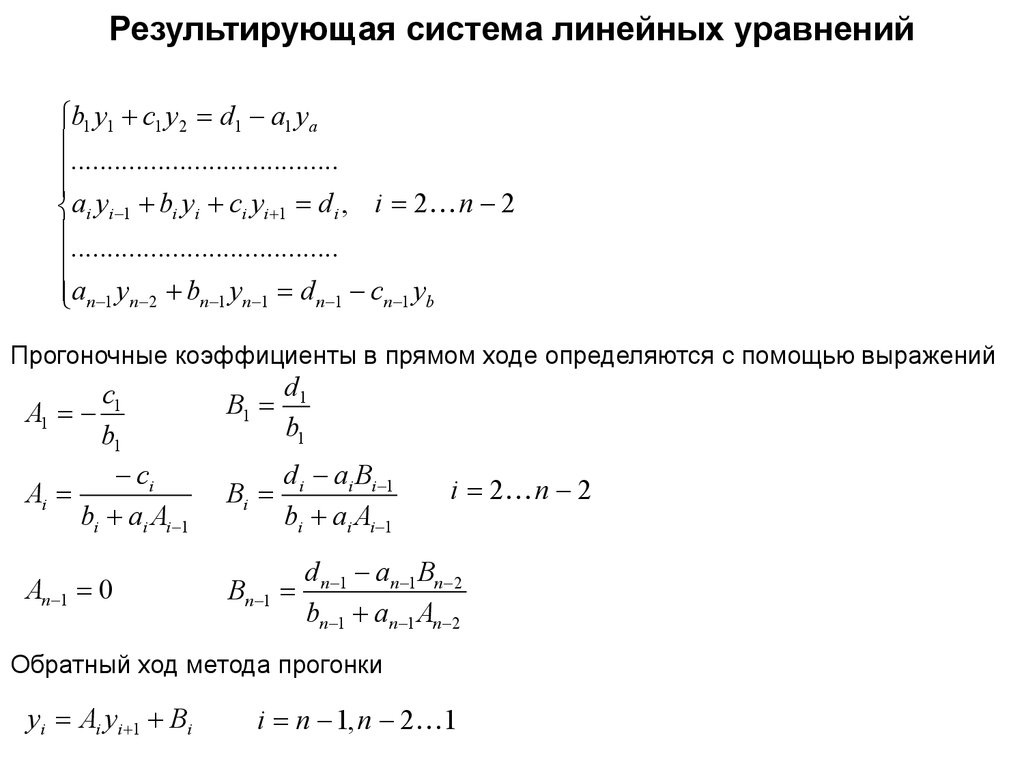
Так как РКЗ представляет собой линейную трехдиагональную систему из *(n+1)* уравнений с *(n+1)* неизвестными, то для ее решения можно применить метод прогонки. Он обладает рядом преимуществ.

Достоинства:

а) число арифметических операций при использовании данного метода составляет *O(n)*, т. е. не превосходит *Kn*  , где  *K= const* .

б) при выполнении условия преобладания диагональных элементов  метод прогонки оказывается слабо чувствительным к ошибкам вычислений: вычислительная погрешность не накапливается с ростом числа узлов *n.*





**Задание 1**

Составить разностную схему и получить численное решение краевой задачи

с точностью *10-3.*

Исходные данные: , где *k* номер варианта().

Граничные условия выбрать однородными

**Решение**

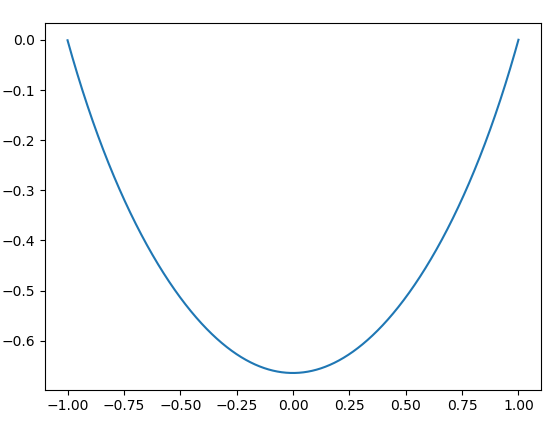
Составим разностную схему:

Заменив в исходном уравнении производные на разностные отношения

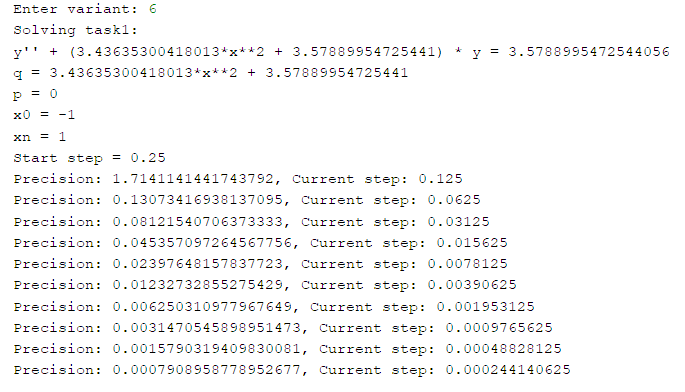
получим:

Решить полученную систему уравнений с трехдиагональной матрицей можно с помощью метода прогонки.

Найдем с помощью метода прогонки решение данной системы и построим график полученной функции при необходимой точности:



Шаг, при котором достигается необходимая точность, равен

Результат работы программы: 

Как видим, при начальном шаге в понадобилось 10 итераций, чтобы добиться необходимой точности.

**Задание 2**

Методом конечных разностей найти приближенное решение краевой задачи:

c точностью ε=0.07 и построить его график. Решение системы разностных уравнений найти, используя метод прогонки.

Граничные условия:

**Решение**

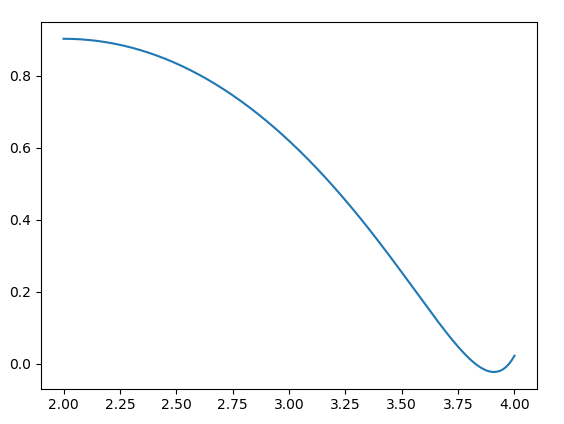
Составим разностную схему:

Заменив в исходном уравнении производные на разностные отношения

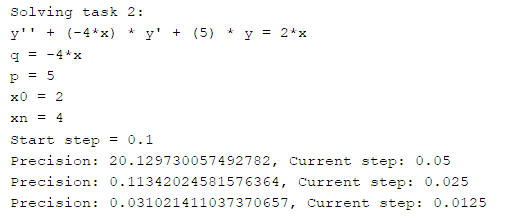
получим:

Решить полученную систему уравнений с трехдиагональной матрицей можно с помощью метода прогонки.

Найдем с помощью метода прогонки решение данной системы и построим график полученной функции при необходимой точности:



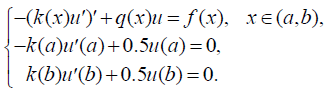
Шаг, при котором достигается необходимая точность, равен

Результат работы программы: 

Как видим, при начальном шаге в понадобилось 3 итерации, чтобы добиться необходимой точности.

**Задание 4**

Методом конечных разностей найти приближенное решение краевой задачи с тремя верными значащими цифрами. Решение системы разностных уравнений:



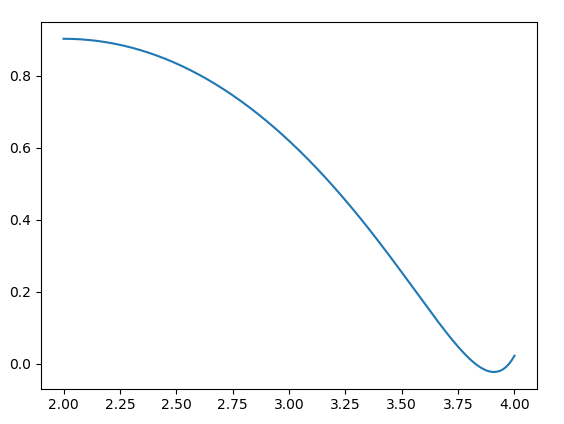
найти, используя метод прогонки.

**Решение**

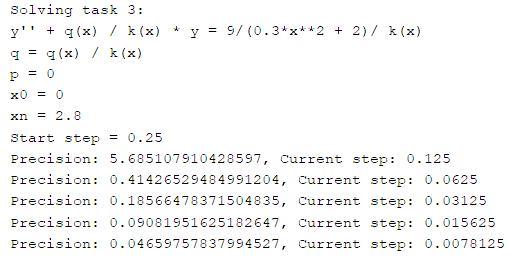
Составим разностную схему:

Решить полученную систему уравнений с трехдиагональной матрицей можно с помощью метода прогонки.

Найдем с помощью метода прогонки решение данной системы и построим график полученной функции при необходимой точности:



Шаг, при котором достигается необходимая точность, равен

Результат работы программы: 

Как видим, при начальном шаге в понадобилось 5 итераций, чтобы добиться необходимой точности.

**Код программы**

**import** numpy **as** np  
**import** sympy **as** sp  
**from** sympy **import** cos, sin, exp  
**import** math  
**from** matplotlib **import** pyplot **as** plt  
  
  
**def** solve\_diag(A, B, C, F):  
 a = [0]  
 b = [0]  
 n = len(A)  
  
 **for** i **in** range(n):  
 a.append(-C[i] / (A[i] \* a[i] + B[i]))  
 b.append((F[i] - A[i] \* b[i]) / (A[i] \* a[i] + B[i]))  
  
 y = np.zeros(n)  
 y[n - 1] = (F[n - 1] - A[n - 1] \* b[n - 1]) / (C[n - 1] + A[n - 1] \* a[n - 1])  
 **for** i **in** range(n - 2, -1, -1):  
 y[i] = a[i + 2] \* y[i + 1] + b[i + 2]  
  
 **return** y  
  
  
**def** f(a):  
 **def** wrapped(x):  
 **return** -1 / a  
 **return** wrapped  
  
  
*# Builds coefficients for diff equation in format y'' - p(x)y = f(x)***def** build\_coeffs\_tridiag(x\_start, x\_end, h, p, g):  
 x = sp.symbols(**'x'**)  
  
 A, B, C, F = [], [], [], []  
 **for** xi **in** np.arange(x\_start, x\_end + h, h):  
 A.append(1)  
 C.append(1)  
 B.append(-h\*\*2 \* p.subs({x: xi}) - 2)  
 F.append(h\*\*2 \* g(xi))  
  
 **return** A, B, C, F  
  
  
**def** build\_coeffs\_tridiag\_task4(x\_start, x\_end, h, p, g):  
 x = sp.symbols(**'x'**)  
  
 A, B, C, F = [], [], [], []  
 **for** xi **in** np.arange(x\_start, x\_end + h, h):  
 A.append(1)  
 C.append(1)  
 B.append(-h\*\*2 \* p(xi) - 2)  
 F.append(h\*\*2 \* g(xi))  
  
 A[0] = 3.6 + h  
 B[0] = -4.8  
 C[0] = 1.2  
 F[0] = F[-1] = 0  
 C[-1] = h + 1.2  
 B[-1] = -1.5  
 A[-1] = 0.4  
  
 **return** A, B, C, F  
  
  
**def** build\_coeffs(x\_start, x\_end, h, p, q, g):  
 x = sp.symbols(**'x'**)  
 n = int((x\_end - x\_start) / h) + 1  
  
 a = np.zeros((n, n))  
 b = np.zeros(n)  
 **for** i, xi **in** enumerate(np.arange(x\_start + h, x\_end, h)):  
 j = i + 1  
 a[j][j - 1] = 2  
 a[j][j] = 2 \* h\*\*2 \* q.subs({x: xi}) - h \* p.subs({x: xi}) - 4  
 a[j][j + 1] = 2 + h \* p.subs({x: xi})  
 b[j] = 2 \* h\*\*2 \* g.subs({x: xi})  
  
 a[0][0] = 2 \* h\*\*2 \* q.subs({x: x\_start}) - 3 \* h \* p.subs({x: x\_start}) - 4  
 a[0][1] = 2 + 4 \* h \* p.subs({x: x\_start})  
 a[0][2] = -h \* p.subs({x: x\_start})  
 b[0] = 2 \* h\*\*2 \* g.subs({x: x\_start})  
  
 a[n - 1][n - 1] = 2 \* h\*\*2 \* q.subs({x: x\_end}) - 3 \* h \* p.subs({x: x\_end}) + 2  
 a[n - 1][n - 2] = -4 - 4 \* h \* p.subs({x: x\_end})  
 a[n - 1][n - 3] = h \* p.subs({x: x\_end})  
 b[n - 1] = 2 \* h \*\* 2 \* g.subs({x: x\_end})  
  
 **return** np.linalg.solve(a, b)  
  
  
**def** solve\_task1(e=0.001):  
 var = int(input(**"Enter variant: "**))  
 a = math.sin(var)  
 b = math.cos(var)  
 p, x = sp.symbols(**"p x"**)  
 p = -(1 + b \* x \*\* 2) / a  
  
 x\_start = -1  
 x\_end = 1  
  
 h = 0.25  
 cur\_precision = 1  
  
 print(**'Solving task1: '**)  
 print(**"y'' + ("** + str(p) + **') \* y = '** + str(-1 / a))  
 print(**'q = '** + str(p))  
 print(**'p = 0'**)  
 print(**'x0 = '** + str(x\_start))  
 print(**'xn = '** + str(x\_end))  
 print(**'Start step = '** + str(h))  
  
 A, B, C, F = build\_coeffs\_tridiag(x\_start, x\_end, h \* 2, p, f(a))  
 y1 = solve\_diag(A, B, C, F)  
 y = []  
  
 **while** cur\_precision > e:  
 y = np.copy(y1)  
 h /= 2  
 A, B, C, F = build\_coeffs\_tridiag(x\_start, x\_end, h, p, f(a))  
 y1 = solve\_diag(A, B, C, F)  
 cur\_precision = np.max(np.abs([y1[2 \* i] - y[i] **for** i **in** range(len(y))]))  
 print(**'Precision: {}, Current step: {}'**.format(cur\_precision, h))  
  
 x1 = np.arange(x\_start, x\_end + h, h)  
 plt.plot(x1, y1)  
 plt.show()  
  
  
**def** solve\_task2(e=0.05):  
 p, q, g, x = sp.symbols(**"p q g x"**)  
 p = cos(x)\*\*2  
 q = 10 / (1 + sin(x)\*\*2)  
 g = exp(-0.5 \* x) \* (12 - x\*\*2)  
  
 x\_start = 0  
 x\_end = 2  
 h = 0.1  
  
 cur\_precision = 1  
 y1 = build\_coeffs(x\_start, x\_end, h, p, q, g)  
 y = []  
  
 **while** cur\_precision > e:  
 y = np.copy(y1)  
 h /= 2  
 y1 = build\_coeffs(x\_start, x\_end, h, p, q, g)  
 cur\_precision = np.max(np.abs([y1[2 \* i] - y[i] **for** i **in** range(len(y))]))  
 print(**'Precision: {}, Current step: {}'**.format(cur\_precision, h))  
  
 plt.plot(np.arange(x\_start, x\_end + h, h), y1)  
 plt.show()  
  
  
**def** build\_coeffs1(x\_start, x\_end, h, p, q, g):  
 x = sp.symbols(**'x'**)  
 n = int((x\_end - x\_start) / h) + 1  
  
 a = np.zeros((n, n))  
 b = np.zeros(n)  
 **for** i, xi **in** enumerate(np.arange(x\_start + h, x\_end, h)):  
 j = i + 1  
 a[j][j - 1] = 2  
 a[j][j] = 2 \* h\*\*2 \* q.subs({x: xi}) - h \* p.subs({x: xi}) - 4  
 a[j][j + 1] = 2 + h \* p.subs({x: xi})  
 b[j] = 2 \* h\*\*2 \* g.subs({x: xi})  
  
 a[0][0] = -3  
 a[0][1] = 4  
 a[0][2] = -1  
 b[0] = 0  
  
 a[n - 1][n - 1] = 2 \* h - 3  
 a[n - 1][n - 2] = 12  
 a[n - 1][n - 3] = -3  
 b[n - 1] = 4 \* h  
  
 **return** np.linalg.solve(a, b)  
  
  
**def** solve\_task3(e=0.07):  
 p, q, g, x = sp.symbols(**"p q g x"**)  
 p = -4 \* x  
 q = 5 + 0 \* x  
 g = 2 \* x  
  
 x\_start = 2  
 x\_end = 4  
 h = 0.1  
  
 print(**'Solving task 2: '**)  
 print(**"y'' + ("** + str(p) + **") \* y' + ("** + str(q) + **') \* y = '** + str(g))  
 print(**'q = '** + str(p))  
 print(**'p = '** + str(q))  
 print(**'x0 = '** + str(x\_start))  
 print(**'xn = '** + str(x\_end))  
 print(**'Start step = '** + str(h))  
  
 cur\_precision = 1  
 y1 = build\_coeffs1(x\_start, x\_end, h, p, q, g)  
 y = []  
  
 **while** cur\_precision > e:  
 y = np.copy(y1)  
 h /= 2  
 y1 = build\_coeffs1(x\_start, x\_end, h, p, q, g)  
 cur\_precision = np.max(np.abs([y1[2 \* i] - y[i] **for** i **in** range(len(y))]))  
 print(**'Precision: {}, Current step: {}'**.format(cur\_precision, h))  
  
 plt.plot(np.arange(x\_start, x\_end + h, h), y1)  
 plt.show()  
  
  
**def** solve\_task4():  
 p, x = sp.symbols(**"p x"**)  
 k1 = 1.2  
 k2 = 0.4  
 q1 = 8.3  
 q2 = 2.8  
 g = 9 / (2 + 0.3 \* x\*\*2)  
 c = 1.875  
 k = **lambda** xi: k1 **if** xi < c **else** k2  
 q = **lambda** xi: q1 **if** xi < c **else** q2  
  
 x\_start = 0  
 x\_end = 2.8  
  
 h = 0.25  
 cur\_precision = 1  
 e = 0.05  
  
 print(**'Solving task 3: '**)  
 print(**"y'' + q(x) / k(x) \* y = "** + str(g) + **"/ k(x)"**)  
 print(**'q = q(x) / k(x)'**)  
 print(**'p = 0'**)  
 print(**'x0 = '** + str(x\_start))  
 print(**'xn = '** + str(x\_end))  
 print(**'Start step = '** + str(h))  
  
 A, B, C, F = build\_coeffs\_tridiag\_task4(x\_start,  
 x\_end,  
 h \* 2,  
 **lambda** xi: -q(xi) / k(xi),  
 **lambda** xi: -g.subs({x: xi}) / k(xi)  
 )  
  
 y1 = solve\_diag(A, B, C, F)  
 y = []  
  
 **while** cur\_precision > e:  
 y = np.copy(y1)  
 h /= 2  
 A, B, C, F = build\_coeffs\_tridiag\_task4(x\_start,  
 x\_end,  
 h,  
 **lambda** xi: -q(xi / k(xi)),  
 **lambda** xi: -g.subs({x: xi}) / k(xi)  
 )  
 y1 = solve\_diag(A, B, C, F)  
 cur\_precision = np.max(np.abs([y1[2 \* i] - y[i] **for** i **in** range(len(y) - 1)]))  
 print(**'Precision: {}, Current step: {}'**.format(cur\_precision, h))  
  
 x1 = np.arange(x\_start, x\_end + h, h)  
 plt.plot(x1, y1)  
 plt.show()  
  
  
**def** main():  
 e = 0.001  
 *solve\_task1(e)  
 solve\_task2()  
 solve\_task3()* solve\_task4()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()