**20K-0264 Assignment 3 BCS-4B  
   
  
Differential Equations:**

**Modify Euler**

# Euler method python program

# function to be solved

def f(x,y):

return x+y

# or

# f = lambda x: x+y

# Euler method

def euler(x0,y0,xn,n):

# Calculating step size

h = (xn-x0)/n

print('\n-----------SOLUTION-----------')

print('------------------------------')

print('x0\ty0\tslope\tyn')

print('------------------------------')

for i in range(n):

slope = f(x0, y0)

yn = y0 + h \* slope

print('%.4f\t%.4f\t%0.4f\t%.4f'% (x0,y0,slope,yn) )

print('------------------------------')

y0 = yn

x0 = x0+h

print('\nAt x=%.4f, y=%.4f' %(xn,yn))

# Inputs

print('Enter initial conditions:')

x0 = float(input('x0 = '))

y0 = float(input('y0 = '))

print('Enter calculation point: ')

xn = float(input('xn = '))

print('Enter number of steps:')

step = int(input('Number of steps = '))

# Euler method call

euler(x0,y0,xn,step)

**4-RK method:**

# RK-4 method python program

# function to be solved

def f(x,y):

return x+y

# or

# f = lambda x: x+y

# RK-4 method

def rk4(x0,y0,xn,n):

# Calculating step size

h = (xn-x0)/n

print('\n--------SOLUTION--------')

print('-------------------------')

print('x0\ty0\tyn')

print('-------------------------')

for i in range(n):

k1 = h \* (f(x0, y0))

k2 = h \* (f((x0+h/2), (y0+k1/2)))

k3 = h \* (f((x0+h/2), (y0+k2/2)))

k4 = h \* (f((x0+h), (y0+k3)))

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

yn = y0 + k

print('%.4f\t%.4f\t%.4f'% (x0,y0,yn) )

print('-------------------------')

y0 = yn

x0 = x0+h

print('\nAt x=%.4f, y=%.4f' %(xn,yn))

# Inputs

print('Enter initial conditions:')

x0 = float(input('x0 = '))

y0 = float(input('y0 = '))

print('Enter calculation point: ')

xn = float(input('xn = '))

print('Enter number of steps:')

step = int(input('Number of steps = '))

# RK4 method call

rk4(x0,y0,xn,step)

**Direct Method for solving linear system:**

**LU decomposition:**

import pprint

import scipy

import scipy.linalg # SciPy Linear Algebra Library

A = scipy.array([ [7, 3, -1, 2], [3, 8, 1, -4], [-1, 1, 4, -1], [2, -4, -1, 6] ])

P, L, U = scipy.linalg.lu(A)

print "A:"

pprint.pprint(A)

print "P:"

pprint.pprint(P)

print "L:"

pprint.pprint(L)

print "U:"

pprint.pprint(U)

**LDLt Factorization :**

def gauss\_solve(A, b):

#Concontanate the matrix A and right hand side column

#vector b into one matrix

temp\_mat = np.c\_[A, b]

#Get the number of rows

n = temp\_mat.shape[0]

#Loop over rows

for i in range(n):

#Find the pivot index by looking down the ith

#column from the ith row to find the maximum

#(in magnitude) entry.

p = np.abs(temp\_mat[i:, i]).argmax()

#We have to reindex the pivot index to be the

#appropriate entry in the entire matrix, not

#just from the ith row down.

p += i

#Swapping rows to make the maximal entry the

#pivot (if needed).

if p != i:

temp\_mat[[p, i]] = temp\_mat[[i, p]]

#Eliminate all entries below the pivot

factor = temp\_mat[i+1:, i] / temp\_mat[i, i]

temp\_mat[i+1:] -= factor[:, np.newaxis] \* temp\_mat[i]

#Allocating space for the solution vector

x = np.zeros\_like(b, dtype=np.double);

#Here we perform the back-substitution. Initializing

#with the last row.

x[-1] = temp\_mat[-1,-1] / temp\_mat[-1, -2]

#Looping over rows in reverse (from the bottom up), starting with the second to

#last row, because the last row solve was completed in the last step.

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

x[i] = (temp\_mat[i,-1] - np.dot(temp\_mat[i,i:-1], x[i:])) / temp\_mat[i,i]

return x

**Iterative methods for solving linear system:**

**Gauss-Siedel:**

def gauss\_seidel(A, b, tolerance=1e-10, max\_iterations=10000):

x = np.zeros\_like(b, dtype=np.double)

#Iterate

for k in range(max\_iterations):

x\_old = x.copy()

#Loop over rows

for i in range(A.shape[0]):

x[i] = (b[i] - np.dot(A[i,:i], x[:i]) - np.dot(A[i,(i+1):], x\_old[(i+1):])) / A[i ,i]

#Stop condition

if np.linalg.norm(x - x\_old, ord=np.inf) / np.linalg.norm(x, ord=np.inf) < tolerance:

break

return x

**Jacobi’s Method:**

def jacobi(A, b, tolerance=1e-10, max\_iterations=10000):

x = np.zeros\_like(b, dtype=np.double)

T = A - np.diag(np.diagonal(A))

for k in range(max\_iterations):

x\_old = x.copy()

x[:] = (b - np.dot(T, x)) / np.diagonal(A)

if np.linalg.norm(x - x\_old, ord=np.inf) / np.linalg.norm(x, ord=np.inf) < tolerance:

break

return x