**Computational Physics PX3143**

**Assignment 1** C1331824



Let and solve differential equation to find lambda :-

Therefore the equation becomes :-

Subbing values into y and to find A and B :-

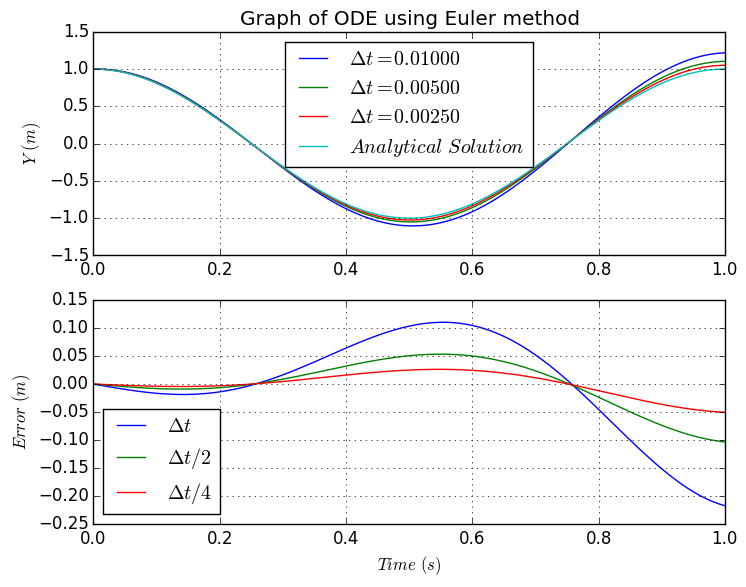
As and

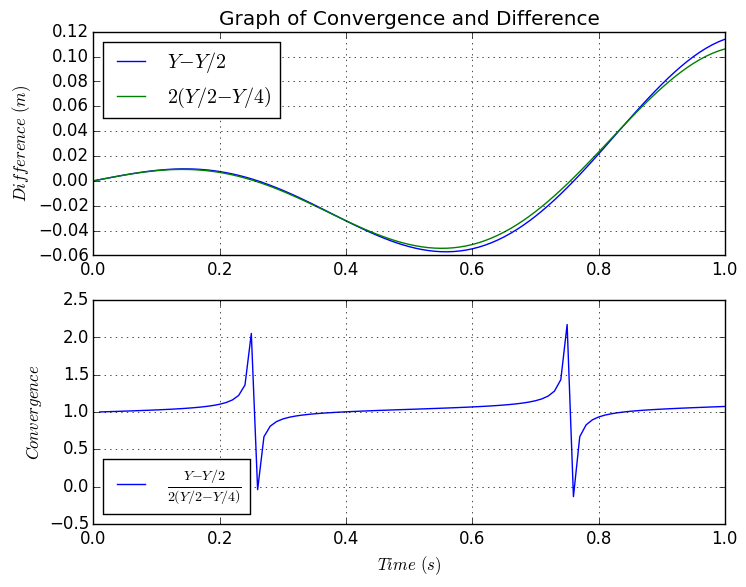
Therefore A = 0.5 and B = 0.5

Which can be reduced into its sine and cosine :-

As

Which leads to the final solution of :-

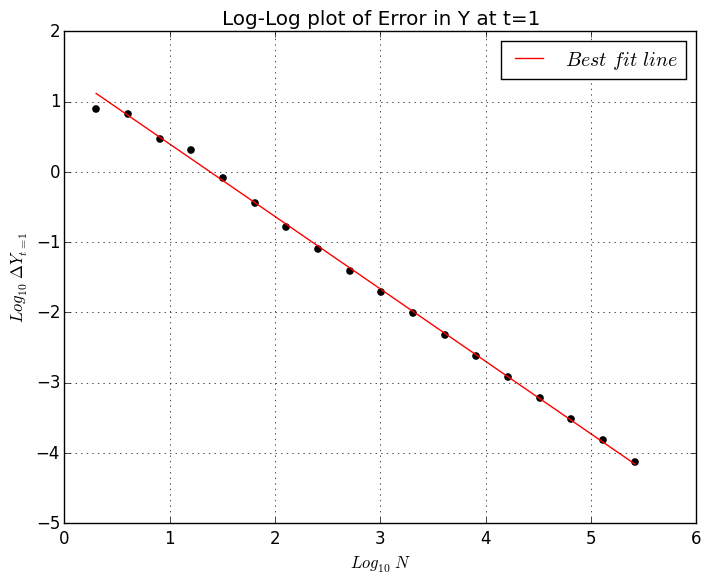
2.

**Figure 1** The above graphs show the numerical solution to the second order differential equation along with difference between the numerical solution and the analytical solution.

**Figure 2** The above graphs show that the error converges at first order, as the line overlaps and that the ratio of the two differences equals one.

The plot of the ratio behaves badly because the Euler method is an unstable method, and that the maximum amplitude of Y-Y/2 is greater than Y/2-Y/4, which causes the two lines to cross, the differences in the maximum amplitudes of these two lines leads to the spikes.

3.



**Figure 3** The above log-log graph shows the error in the Y value at for 18 different time steps, where

4.

As the number of integration steps increases, the error in Y decreases which agrees with previous graphs (Figure 1) that show that as dt decreases the closer the numerical solution is to the analytical solution.

As the straight line fit closely overlaps the data points it can be said that the code is first-order convergent. Above integration steps the code becomes first-order convergent.

Below where the convergence is unclean, because due to the stability of Euler method (), dt must be less than 2/lambda (), where λ is equal to 4π2. Using the equation, we can see that dt must be less than ~0.05 and therefore N must be greater than ~20 which roughly agrees with my value of (N = 30), therefore below this value of N = 20 the method becomes unstable which can be seen in the graph.

**Double click on Assignment 1 for file**

# -\*- coding: utf-8 -\*-

"""

Created on Tue Oct 06 09:37:15 2016

@author: C1331824

"""

from \_\_future\_\_ import division

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import curve\_fit

def f(y, l):

""" A function to call yprime and y """

return y[1], l\*y[0]

def Euler(ya, f, dt, l):

""" The Y array has two values so we need to split them up

when using euler """

fy,fyp = f(ya, l) # fy gives the value y of the function and fyp gives the value of yprime of the function

yi,ypi = ya # yi gives the initial y values and ypi gives initail yp values

# using euler method

y = yi + dt\*fy # y value

yp = ypi + dt\*fyp # yprime value

return y, yp

def ODEsolve(Tmax, N, f, method, ic):

""" A function to do all of the ODE in one bit """

t = np.zeros(N+1) # defining the time array, adding 1 to get the time array to go to tmax and not Tmax - dt like it was doing before

dt = Tmax/N; t[0] = ic[0]

y = np.zeros((2,N+1)) # defining a y array containging the y values and yp values and + 1 to keep the size of Y and T consistent

y[0,0] = ic[1]; y[1,0] = ic[2]

for i in range(0,int(N)):

y[:,i+1] = method(y[:,i], f, dt , ic[3]) # doing the euler method to get both y and yprime

t[i+1] = t[i] + dt

return y, t

# lambda

w = 2\*np.pi; l = -w\*w # defining my constants

# defining initial conditions

ti = 0; Tmax = 1 # starting time = 0, final time = 1

yi = 1; ypi = 0 # starting y = 0 and yprime = 1

# time steps

N = 100; n = np.array([N,2\*N,4\*N])

# collecting initial conditions

ic = np.array([ti, yi, ypi, l]) # initial time, initial y, initial yprime and lambda

# solving ODE

R = [ODEsolve(Tmax, N, f, Euler, ic) for i,N in enumerate(n)]

# defining analytic solution

def f1(t,l):

return np.cos(l\*t) # exact solution of d^2y/dt^2

F = [r'$\Delta t$',r'$\Delta t/2$',r'$\Delta t/4$'] # used for labeling graphs

for i in range(len(R)): # for loop to do all three step sizes at once

T = R[i][1]; Y = R[i][0][0]

plt.subplot(2,1,1)

plt.plot(T, Y, label=r'$\Delta t = %.5f$' %(Tmax/n[i])) # ploting my Y solutions

plt.ylabel(r'$Y$ $(m)$')

plt.title('Graph of ODE using Euler method')

plt.legend(loc='best')

plt.xlim(0,1)

plt.grid()

plt.subplot(2,1,2)

plt.plot(T, f1(T,w) - Y, label=F[i] ) # plotting the difference between the numerical and analytical solutions

plt.xlabel(r'$Time$ $(s)$')

plt.ylabel(r'$Error$ $(m)$')

plt.legend(loc='best')

plt.xlim(0,1)

plt.grid()

plt.subplot(2,1,1)

plt.plot(T,f1(T,w), label=r'$Analytical$ $Solution$')

plt.legend(loc='best')

plt.savefig('Graph of ODE.png', bbox\_inches='tight')

# creating my self-convergence test

def ConvergenceTest(ODEsolve, Tmax, n, f, ic, method, order):

""" Creating a self-convergence test to see for correct order """

R = [ODEsolve(Tmax, N, f, method, ic) for i,N in enumerate(n)]

Y1 = R[0][0][0]; Y2 = R[1][0][0]; Y4 = R[2][0][0]

diff1 = (Y1 - Y2[::2]) # [::2] skips every other indices in the array and 4 skips every fourth to have the same size arrays to be able to take the differences

diff2 = (2\*\*order)\*(Y2[::2] - Y4[::4])

return diff1,diff2

d1,d2 = ConvergenceTest(ODEsolve, Tmax, n, f, ic, Euler, 1) # where d1 is Ydt - Ydt/2 and d2 is 2\*Ydt/2 - Ydt/4

# generating the ratio of differences

ratio = d1/d2

plt.figure()

plt.subplot(2,1,1)

plt.plot(T[::4],d1, label=r'$Y - Y/2$') # plotting Ydt - Ydt/2 and 2\*Ydt/2-Ydt/4 to show that euler is first order convergent

plt.plot(T[::4],d2, label=r'$2\left(Y/2 - Y/4\right)$')

plt.ylabel(r'$Difference$ $(m)$')

plt.title('Graph of Convergence and Difference')

plt.legend(loc='best')

plt.grid()

plt.subplot(2,1,2)

plt.plot(T[::4],ratio, label=r'$\frac{Y - Y/2}{2\left(Y/2 - Y/4\right)}$') # plotting Ydt - Ydt/2 / 2\*Ydt/2-Ydt/4 to show that euler is first order convergent

plt.xlabel(r'$Time$ $(s)$')

plt.ylabel(r'$Convergence$')

plt.legend(loc='best')

plt.grid()

plt.savefig('Graph of Convergence.png', bbox\_inches='tight')

# creating my 18 array of dt steps

dt = (1/2)\*\*np.linspace(1,18,18); Na = Tmax/dt

A = [ODEsolve(Tmax, n, f, Euler, ic) for i, n in enumerate(Na)] # creating 18 different arrays to be able to get the final point in each array

Ydt = [A[i][0][0][-1] for i in range(len(A))] # get the final point in each array

Tdt = [A[i][1][-1] for i in range(len(A))]

Te = np.array(Tdt); Ye = f1(Te,w); Yt = -np.array(Ydt) # to make sure all values are positive so not to get errors when taking logs of negative values

LN = np.log10(Na); LY = np.log10(Ydt - Ye); LMY = np.log10(Yt - Ye) # taking logs of the step sizes and the difference between analytical and numerical solution

def bestfit(x, m, c):

""" Creating a best fit line to see the range in which

the code is first-order convergent """

return m \* x + c

# getting my best fit line

popt, pcov = curve\_fit(bestfit, LN[2:], LY[2:])

YB = bestfit(LN, popt[0], popt[1]) # best fit line to see at what range is code first order convergent

fig = plt.figure()

ax1 = fig.add\_subplot(1,1,1)

ax1.scatter(LN[2:], LY[2:], c = 'k') # plot of loglog steps v differences

ax1.scatter(LN[:2], LMY[:2], c = 'k')

ax1.plot(LN, YB, c = 'r', label=r"$Best$ $fit$ $line$") # plot of best fit line

plt.xlabel(r'$Log\_{10}$ $N$')

plt.ylabel(r'$Log\_{10}$ $\Delta Y\_{t=1}$')

plt.title('Log-Log plot of Error in Y at t=1')

plt.legend(loc='best')

plt.grid()

plt.show()

plt.savefig('Graph of Error.png', bbox\_inches='tight')