John Anderson
HW6
MECE 6397

3) Computational

Part a)

Starting with 10 interior grid and subdividing the total ten interior time steps, the average error at T=10 is already only 0.0055. Doubling the number of interior grid points just once gives an error of 0.0011 and makes a graph of the exact and approximate values indistinguishable without zooming in multiple times. Doubling the grid points again from 20 to 40 gives an error of 0.00019. Doubling the grid points from 40 to 80 actually increases the error slightly. At this point further increasing the number of internal grid points increases the error as we have error propagation occurring at each step.

The error in a Crank Nicolson scheme does not depend only on the size delta x, but also on delta t. The error is on the order of (delta x^2 +delta t ^2). By keeping the number of interior x points constant and doubling the number of time points we the same trend that we saw before. The error decreases as delta t becomes smaller until the effect of error propagation takes over. Interestingly enough the error for 10 interior grid points with 20-time steps is greater than 10 interior grid points and 10 interior time points, as the error from 10 times steps is now propagated 20 times rather than 10.

Just to examine the error for decreasing dx and dt I put part of my python code in a double for loop and recorded the error for doubling the number of interior points and the number of time points. This was a little excessive, but I wanted to examine the error and try to find the "best error", as in what is the smallest error fewest number of total points, while also considering how long it takes to run on my personal laptop. A portion of the results of this exercise are presented in Table 1. The full results are available in the excel file contained in GitHub.

Table 1. Average error part a for increasing the number of interior time and x points.

Average Error at t=T for part a							
NT=number of interior t points							
N=number of interior x points							
	NT=10	NT=20	NT=40	NT=80	NT=160	NT=320	NT=640
N=10	5.510E-03	5.954E-03	6.078E-03	6.110E-03	6.119E-03	6.121E-03	6.121E-03
N=20	1.120E-03	1.593E-03	1.725E-03	1.760E-03	1.769E-03	1.771E-03	1.772E-03
N=40	1.944E-04	2.930E-04	4.288E-04	4.647E-04	4.739E-04	4.763E-04	4.769E-04
N=80	5.567E-04	6.281E-05	7.485E-05	1.113E-04	1.206E-04	1.230E-04	1.236E-04
N=160	6.534E-04	1.563E-04	1.773E-05	1.891E-05	2.834E-05	3.073E-05	3.133E-05
N=320	6.792E-04	1.805E-04	4.146E-05	4.705E-06	4.752E-06	7.151E-06	7.755E-06
N=640	6.862E-04	1.867E-04	4.750E-05	1.068E-05	1.211E-06	1.191E-06	1.796E-06

We can see that moving diagonally across the table always results in a lower average error, as delta x and delta t have both decreased. However, when moving horizontally across a row or vertically down a column there is a point where the average error increases. The error propagation can for both the x and t dimension and eventually over comes the benefit from

further decreasing Δx or Δt. The smallest average error calculated was 1.84E-08 and used 5120 interior x points and 2150 interior time points. The "best error" from the table occurs with 80 interior grid points for x and 20 interior points for time, as it is the first error in the table reach an average error on the order of E-05. We can also see at this value doubling only the interior x points or t points results in a larger average error. At this point we must double both the x and t points to further reduce the error.

Using Table 1 as a guide I moved diagonally down the table starting from 80 interior grid points and 20 interior time points. I used time functions in python to measure how long that portion of the code took to ran. 320 interior x points and 80 interior t points took under one second to run, while 640 interior x points and 160 interior t points t took over one second to run. Using 1 second as an arbitrary cutoff I chose to use the 320 interior x points and 80 interior t points as the most accurate results presented in this report. Table 2 compares just a few values for the exact and approximate solution using 10 interior points for x and t. Table 3 compares just a few values for the exact and approximate solution using 320 interior points for x and 80 interior points t. The largest error occurs near the center of the interval and at the last time step as we get further away from the Dirichlet boundary conditions.

Table 2. Comparison of exact and approximate solutions at selective values using 10 interior points for x and t.

10 Interior x Points, 10 Interior t Points				
Average Error =5.51E-03				
u_appx				
	x=0.2856	x=0.5712	x=1.4280	x=2.8560
t=0.9091	0.2574	0.4939	0.9043	0.2574
t=1.8182	0.2352	0.4513	0.8262	0.2352
t=9.091	0.1141	0.2190	0.4010	0.1141
t=10	0.1043	0.2001	0.3664	0.1043
u_exact				
	x=0.2856	x=0.5712	x=1.4280	x=2.8560
t=0.9091	0.2573	0.4937	0.9038	0.2573
t=1.8182	0.2349	0.4508	0.8253	0.2349
t=9.091	0.1135	0.2178	0.3988	0.1135
t=10	0.1036	0.1989	0.3641	0.1036

Table 3. Comparison of exact and approximate solutions at selective values using 320 interior points for x and 80 interiors.

320 Interior x Points, 80 Interior t Points				
Average Error=4.70E-06				
u_appx				
	x=0.0098	x=1.5757	x=2.5054	x=3.1318
t=0.1235	0.009666655	0.987718333	0.586811852	0.009666655
t=0.247	0.009548047	0.975599186	0.579611764	0.009548047
t=9.8765	0.003645046	0.372443000	0.221271621	0.003645050
t=10	0.003600322	0.367873000	0.218556653	0.003600320
u_exact				
	x=0.0098	x=1.5757	x=2.5054	x=3.1318
t=0.1235	0.009666656	0.987718390	0.586811886	0.009666656
t=0.247	0.009548048	0.975599299	0.579611831	0.009548048
t=9.8765	0.003645063	0.372444854	0.221272652	0.003645063
t=10	0.003600339	0.367875037	0.218557684	0.003600339

Figure 1 is a graph of the exact and approximate solution using interior points for x and t. The exact and approximate solutions are already very close. Figure 2 is for 20 interior x points and 10 interior t points and the two lines are indistinguishable from one another. Graphs at different times, like T/5 for example, appear the exact same but with a different maximum value that the parabola reaches. A more useful figure is obtained by creating a 3d surface plot showing how the function values change in time and space. We can see in Figure 3 the function is parabolic and the parabola decreases in height as time increases. This is the effect of the exponential term in the exact solution, which scales the sine curve as time increases.

John Anderson
HW6
MECE 6397

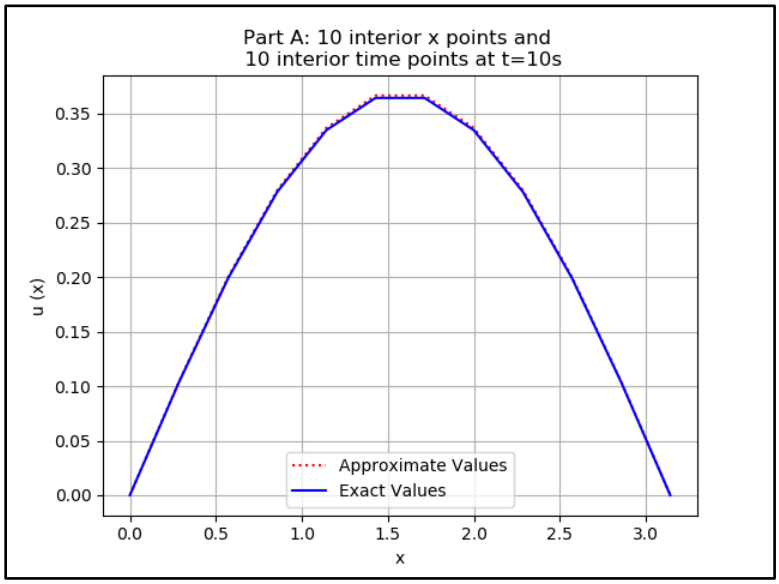

Fig 1. Graph of approximate and exact solution at t=T. 10 interior x points and 10 t points.

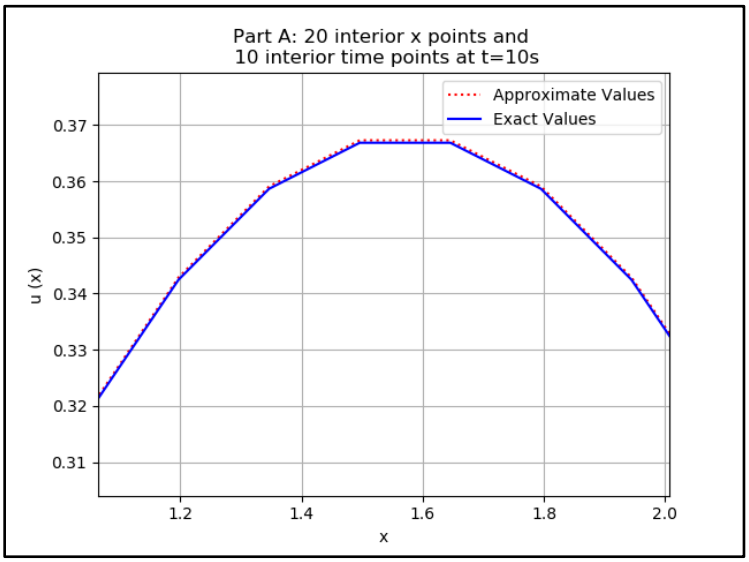

Fig. 2. Zoomed in graph of approximate and exact solution at t=T. 20 interior x points and 10 t points. The lines are indistinguashable from eachother without zooming in.

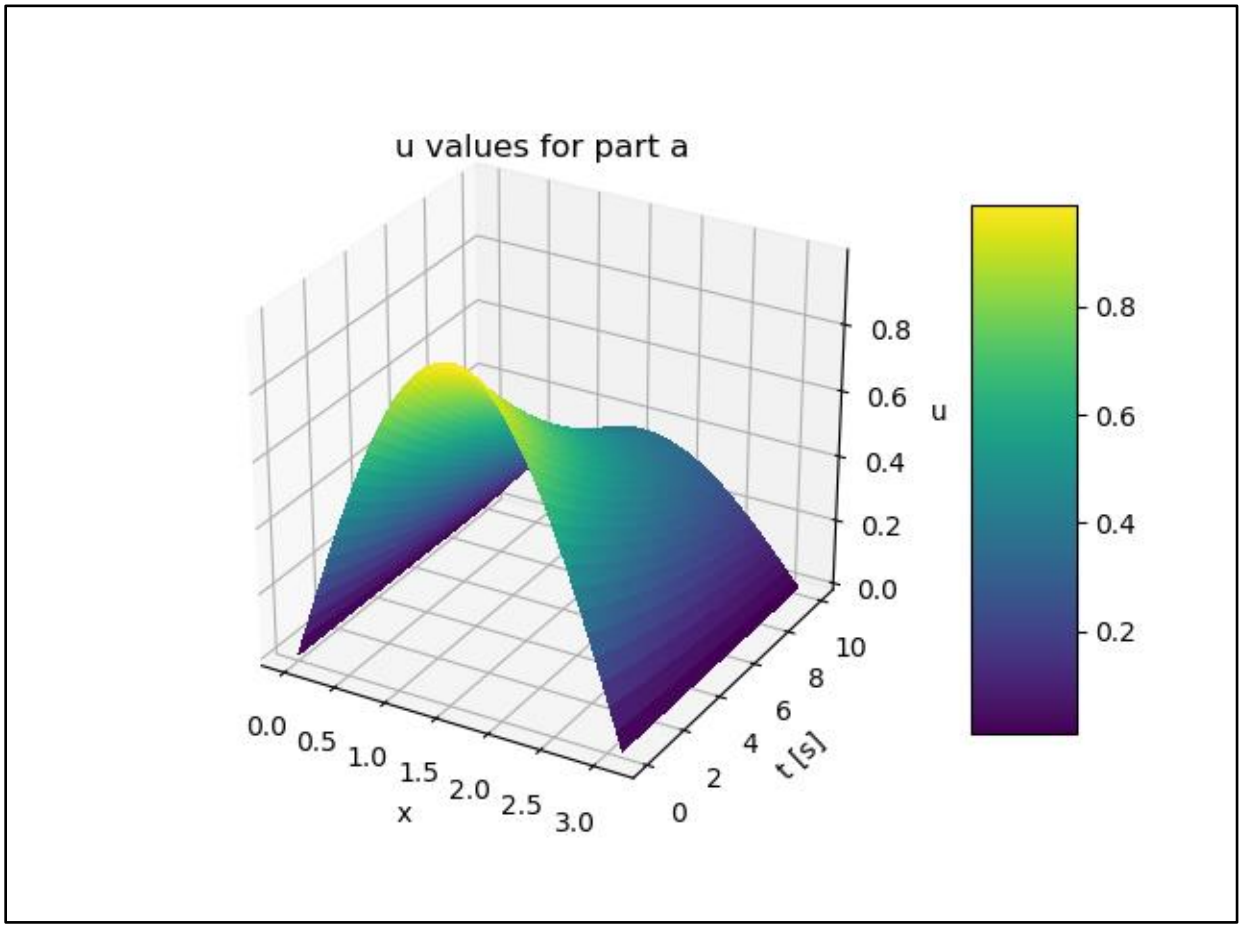

Fig 3. 3D surface plot of the function showing the how the values change in space and time.

John Anderson
HW6
MECE 6397

Part B)

Starting with 10 interior grid and subdividing the total time into one second intervals (ten interior time steps), the average error at T=10 is already only 0.0013 and a graph of the approximate and exact solution is indistinguishable form one another with out zooming in multiple times. We can double the number of x points twice to 40 and reach an average error 7.05E-05. Doubling the number of interior x points again increases the error as the effect of propagation outweighs the further decrease in delta x. Figures 4 and 5 below show zoomed in graphs of the exact and approximate solutions at t=T. Tables 4 and 5 compare selected values for the exact and approximate solutions. Figure 5 is a 3d graph to show how the solution changes in time and space.

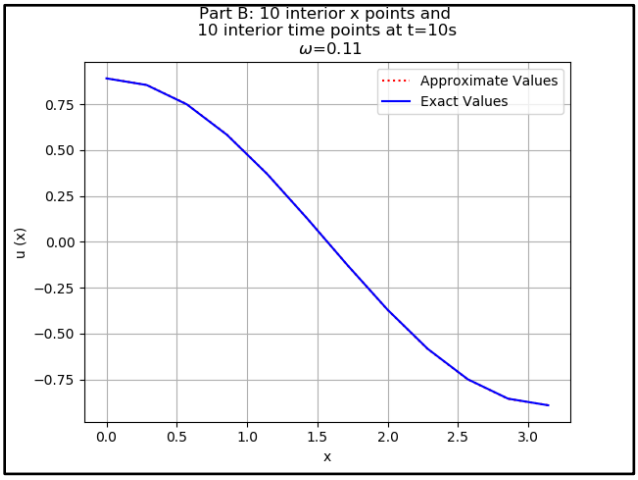

Fig 3. Graph of approximate and exact solution at t=T. 10 interior x points and 10 t points.

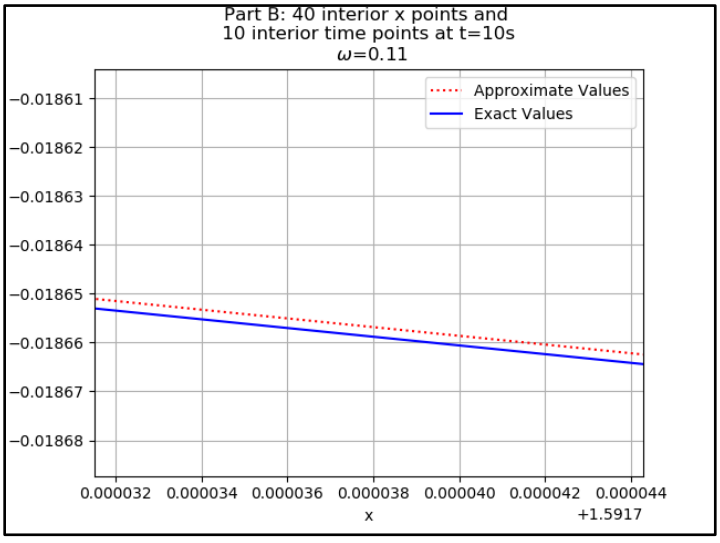

Fig. 4. Zoomed in graph of approximate and exact solution at t=T. 40 interior x points and 10 t points. The lines are indistinguashable from eachother without zooming in.

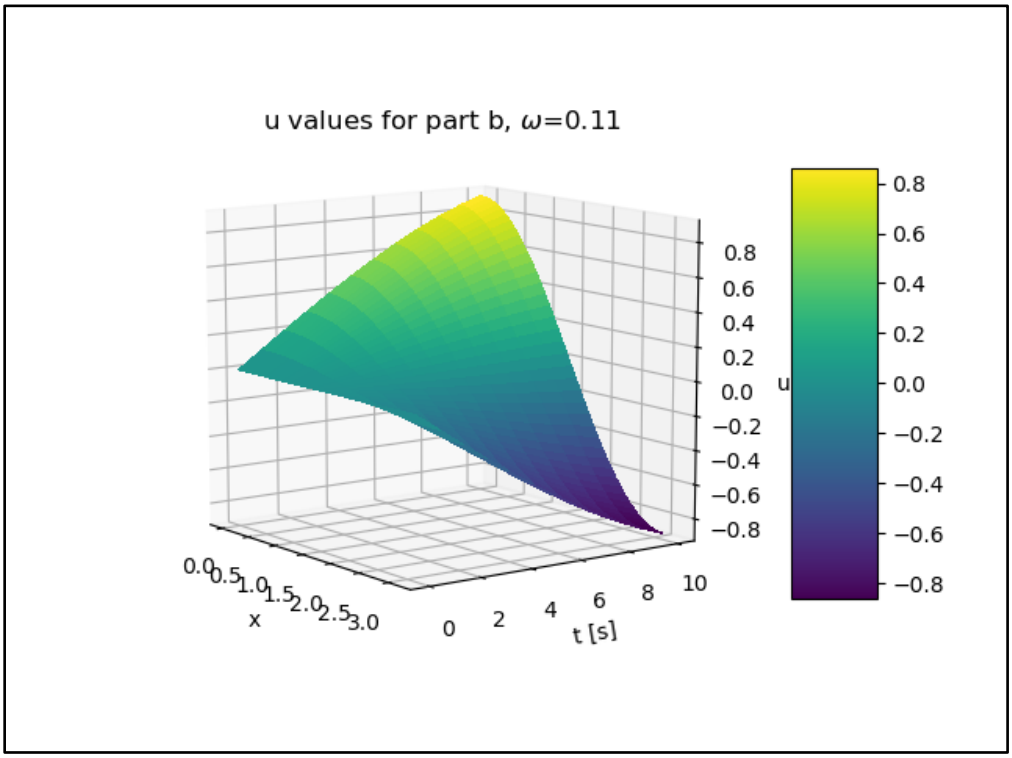

Fig. 5 3d surface plot for 40 interior x points, 10 interior t points.

Table 4. Selected table values using 10 interior points for x and t.

10 Interior x Points, 10 Interior t Points				
Average Error =1.25E-03				
u_appx				
	x=0.2856	x=0.5712	x=1.4280	x=2.8560
t=0.9091	0.0958	0.0839	0.0142	-0.0958
t=1.8182	0.1906	0.1671	0.0283	-0.1906
t=9.091	0.8079	0.7087	0.1199	-0.8079
t=10	0.8557	0.7506	0.1270	-0.8557
u_exact				
	x=0.2856	x=0.5712	x=1.4280	x=2.8560
t=0.9091	0.0958	0.0840	0.0142	-0.0958
t=1.8182	0.1906	0.1671	0.0283	-0.1906
t=9.091	0.8074	0.7079	0.1198	-0.8074
t=10	0.8551	0.7497	0.1268	-0.8551

Table 5. Table 4. Selected table values using 10 interior points for x and t.

40 Interior x Points, 10 Interior t Points				
Average Error=7.05E-05				
u_appx				
	x=0.0766	x=0.1532	x= 1.5325	x=3.0650
t=0.9091	0.09952	0.09862	0.00382	-0.09952
t=1.8182	0.19806	0.19630	0.00760	-0.19806
t=9.091	0.83899	0.83159	0.03223	-0.83899
t=10	0.88858	0.88075	0.03413	-0.88858
u_exact				
	x=0.0766	x=0.1532	x= 1.5325	x=3.0650
t=0.9091	0.09954	0.09866	0.00382	-0.09954
t=1.8182	0.19809	0.19634	0.00761	-0.19809
t=9.091	0.83900	0.83161	0.03223	-0.83900
t=10	0.88859	0.88076	0.03414	-0.88859

Because of the way the omega value is defined in this problem we can not increase the number of time points like we did in part a without changing the actual function itself. We are instead to asked vary omega for a fixed Δt, starting with $\omega\Delta t=0.1$ and increasing on up. This also changes the value of the function, the boundary conditions, and the prescribed function, so that essentially we are examining the error for a different problem each time. The issue here is that as we increase omega, which is the angular frequency, while keeping Δt the same we are missing out on the behavior of the function. We can take this to the extreme and imagine a function that oscillates 1 million times during our 10 second interval. Clearly only 10 interior times is inadequate to accurately approximate this function. Table 6 presents the average error for varying angular frequency and includes one absurd case with error and frequency on the order of E55.

Table 6. Average error for different angular frequencies.

Average Error at t=T for Increasing Angular Frequency	
10 interior x points and 10 interior t points	
Angular Frequency rad/s	Average Error
0.11	0.001251969
1.1	0.076130902
11	1.897405018
110	28.16645906
1100	909.5416217
1.10E+55	5.26E+55

Note: k=1 was used for all equations in this assignment.

John Anderson
HW6
MECE 6397

Python Code

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov  5 18:11:56 2020

@author: johna
"""

# MECE 6397, SciComp, HW 6, Computational
# 1-D Diffusion Problem Crank Nicolson
#https://github.com/jeander5/MECE_6397_HW6_COMP

#imports
import math
import numpy as np
import matplotlib.pyplot as plt
from math import sin as sin
from math import cos as cos
#from mpl_toolkits import mplot3d

#Constants given in problem statement, constant for both boundary conditions
L=math.pi
D=0.1
T=10
#k is just an integer
k=1

#discretize the interval function, also works for time
def DIF(L ,N):
    """discretizes an interval with N interior points"""
```

```
#Inputs are interval lengths number of interior points
    h = L/(N+1)
    x = np.linspace(0, L, N+2)
    return(x, h)


def thomas_alg_func(a,b,c,f):
    """solves tridiagonal matrix"""
#inputs are vectors containing the tridiagonal elements and right hand side
    N=len(a)
#storage vectors
    u_appx = [0]*N
    alpha = [0]*N
    g = [0]*N
#Following the pseudocode
#Zeroth element of this list corresponds to the first subscript in Thomas Algorithm
    alpha[0] = a[0]
    g[0] = f[0]
    for j in range(1, N):
        alpha[j] = a[j]-(b[j]/alpha[j-1])*c[j-1]
        g[j] = f[j]-(b[j]/alpha[j-1])*g[j-1]
    u_appx[N-1] = g[N-1]/alpha[N-1]
    for j in range(1, N):
        u_appx[-1-j] = (g[-1-j]-c[-1-j]*u_appx[-j])/alpha[-1-j]
    return u_appx


#u exact function part a
def u_exact_func_a(k, D, t, x):
    """returns exact u values for the function from Part a"""
#Inputs are x and t values, and the given constants
    len_x = len(x)
```

```
    len_t = len(t)
    func_vals=np.zeros(shape=(len_t, len_x))
    for n in range(len_t):
        func_vals[n,1:-1]=[math.exp(-D*k*k*t[n])*sin(k*x) for x in x[1:-1]]
#note I have from x in x[1:-1] in here to just keep the values as zero so I dont need to reassign them
# the boundary conditions u(x=0,t) and  u(x=L,t) are zero for all t
    return func_vals


#u exact function part b
def u_exact_func_b(k, D, w, t, x):
    """returns exact u values for the function from Part b"""
#Inputs are x and t values, and the given constants
    len_x = len(x)
    len_t = len(t)
    func_vals=np.zeros(shape=(len_t, len_x))
    for n in range(len_t):
        func_vals[n,:]=[sin(w*t[n])*cos(k*x) for x in x]
    return func_vals


#dont really need functions for these but I made them.
#it is nice to have so I can change the inputs on the fly
def BC_Partb(t,w,k,L):
    """returns boundary conditions for the function from Part b"""
#inputs are the time list and the given constants
    a=cos(k*L)
    g_0b =[sin(w*t) for t in t]
    g_Lb= [sin(w*t)*a for t in t]
    return(g_0b, g_Lb)
```

```
#The prescribed function F
def preF(D,k,w,x,t):
    """returns values for the prescribed function F(x,t)"""
#Inputs are x and t values, and the given constants
    len_x = len(x)
    len_t = len(t)
    func_vals=np.zeros(shape=(len_t, len_x))
    for n in range(len_t):
        func_vals[n,:]=[cos(k*x)*((w*cos(w*t[n])+D*k*k*sin(w*t[n]))) for x in x]
    return func_vals


def avg_error (exact,appx):
    """returns average error"""
#inputs are just single row for exact and approximate solution
#the entire x domain at a single time
    N=len(exact)-2
    mysum=0
    for j in range(1,N):
        mysum=mysum+abs((appx[j]-exact[j])/exact[j])
    error=mysum/N
    return error


# N grid points for x
##NT is  "grid points" for t,
N=40
NT=10


#calling the DIF
x, dx = DIF(L,N)
t, dt = DIF (T,NT)
```

```
        #lets define omega right here , after dt is defined
        #for a fixed dt, try increasing omega,
        w=0.1/dt

        #lengths defined here so they dont need to be multiple times later
        len_x = len(x)
        len_t = len(t)

        #constants from the CN scheme used in the thomas algorithm
        lamda=D*dt/(dx*dx)
        b=-lamda/2
        a=1+lamda
        c=-lamda/2
        #im defining a d so it doesnt have to calculate inside loops or functions
        d=1-lamda

        #
=========================================================================
==
        # part A
        #
=========================================================================
==

        #solution matrix
        u_appx_a=np.zeros(shape=(len_t, len_x))

        #Assigning Intital Conditions
        u_appx_a[0,:]=[sin(k*x) for x in x]

        #input vecotrs for the thomas algorithm function
```

```
av= [a]*N
bv= [b]*N
cv= [c]*N
rhs= [0]*N

#for loop for calling the thomas algorithm at the different time steps
for n in range (1,len_t):
#q is just aplaceholder variable it just makes the code a little cleaner
   q=u_appx_a[n-1,:]
# first input different
   rhs[0]=-b*q[0]+d*q[1]-c*q[2]-b*u_appx_a[n,0]
#    inner for loop for filling up the rhs vector for the thomas algorithm
   for j in range (1,N-1):
      rhs[j]=-b*q[j]+d*q[j+1]-c*q[j+2]
   #last input different
      j=j+1
      rhs[j]=-b*q[j]+d*q[j+1]-c*q[j+2]-c*u_appx_a[n,-1]
   u_appx_a[n,1:-1]=thomas_alg_func(av,bv,cv,rhs)

#calling exact function
u_exact_a=u_exact_func_a(k, D, t, x)

#calling the error function
error_a=avg_error(u_exact_a[-1],u_appx_a[-1])

#
==========================================================================
==
# part B
```

```
        #
==========================================================================
==


        #values given in problem statement
        g_0,g_L=BC_Partb(t,w,k,L)
        #initial condition
        f=[0]*len_x
        #Prescibed function F
        F=preF(D,k,w,x,t)

        #solution matrix, will be filled in
        u_appx_b=np.zeros(shape=(len_t, len_x))

        #Boundary Conditions
        u_appx_b[:,0]=g_0
        u_appx_b[:,-1]=g_L

        #Initial Conditions
        u_appx_b[0,1:-1]=f[1:-1]
        #I still have [1:-1] no need to reassign those endpoints for t =0

        #for loop for calling the thomas algorithm at the different time steps
        #input vecotrs for the thomas algorithm function are already defined from part a
        #I am just redefining the rhs vector inputs
        rhs= [0]*N
        for n in range (1,len_t):
        #from 1, len_t because initial conditions are defined
        #Big Q and little q are just place holders. it just makes the code a little cleaner
           Q=np.array(1/2*(F[n-1,:]+F[n,:]))
```

```
    q=u_appx_b[n-1,:]


#first input of rhs different
    rhs[0]=-b*q[0]+d*q[1]-c*q[2]-b*u_appx_b[n,0]+dt*Q[1]
#    inner for loop for filling up the rhs vector for the thomas algorithm
    for j in range (1,N-1):
        rhs[j]=-b*q[j]+d*q[j+1]-c*q[j+2]+dt*Q[j+1]
#last input of rhs different
    j=j+1
    rhs[j]=-b*q[j]+d*q[j+1]-c*q[j+2]-c*u_appx_b[n,-1]+dt*Q[j+1]
    u_appx_b[n,1:-1]=thomas_alg_func(av,bv,cv,rhs)


#calling exact function
u_exact_b=u_exact_func_b(k, D, w, t, x)
#calling the error function
error_b=avg_error(u_exact_b[-1],u_appx_b[-1])


#
================================================================================
==
# plotting
#
================================================================================
==
#uncomment these as needed for the report
#fig, ax = plt.subplots()
#plt.grid(1)
#plt.plot(x,u_appx_b[-1],'r:')
#plt.plot(x,u_exact_b[-1],'b')
#ax.legend(['Approximate Values','Exact Values'])
##ax.title(['Part A: %s interior grid points and a time interval of'%(N)])
```

```
#ax.title.set_text(
#       'Part B: %s interior x points and \n %s interior time points at t=%ss \n $\omega$=%s'
#               %(N,NT,T,round(w,3)))
#ax.set_xlabel('x')
#ax.set_ylabel('u (x)')



#3d graphing im only gonna keep the surface one
#fig2 = plt.figure()
#ax = plt.axes(projection='3d')
#BIGX, BIGT = np.meshgrid(x, t)
#from matplotlib import cm
#surf = ax.plot_surface(BIGX, BIGT, u_appx_b, cmap=cm.viridis,
#                       linewidth=0, antialiased=False)
#fig2.colorbar(surf, shrink=0.75, aspect=5)
#ax.set_title('u values for part b, $\omega$=%s'%(round(w,3)))
#ax.set_xlabel('x')
#ax.set_ylabel('t [s]')
#ax.set_zlabel('u')
```