MECE 6397

Graduate Student Project

John Anderson

Fall 2020

John Anderson
MECE 6397
Grad Student Project

Table of Contents

John Anderson
MECE 6397
Grad Student Project

Introduction

The problem to be solved was a 2D diffusion one with no forcing function. The domain of interest with was square with side lengths equal to 2π. Dirichlet boundary condition were imposed on the top, bottom, and right side of the domain. The Dirichlet boundary condition were all scaled by the quantity $(1 - e^{-\varphi t})$, which caused them to start at zero and eventually reach a steady value. A Neumann Boundary condition was specified on the left side of the domain. The initial condition was prescribed as $u(x, y, t) = 0$.

A computer code was written to find the steady state solution to this problem. The coding language chosen was Python. The Alternating Direction Implicit (ADI) scheme was used as well as the explicit scheme. The ghost node method was implemented for the Neumann the boundary condition. The code is segmented with different functions for boundary conditions, tridiagonal solver, error functions, etc. The code was kept flexible and one can change parameters and run the code for a different problem. Inevitably some aspects of the code are specific to this problem, but the code is easily changeable. For example, if the Neumann boundary condition was on the opposite side some modifications in the ADI function would be needed, but these would be easily implemented. Indeed, this code was easily changed when the updated problem statement was received, which made the Dirichlet boundary conditions functions of time as well. The code was written to be efficient and makes use of some of the coding techniques we have learned throughout the term. For example, terms that can be calculated outside loops are to save on calculation time. There was a tradeoff between efficiency and adaptability. There are some instances in the code where a value of zero is added to an expression, as a result of the specifics of this problem. These terms were left in the code to be as general as possible. The code is comment and GitHub is used for version control.

The basic method for solving this problem was to advance the solution forward in time and compare it to the solution at the previous time step. If the maximum difference between the $u(x, y, t^n)$ and $u(x, y, t^{n+1})$ was less than some prescribed value, then the steady state solution is reached. To avoid coding inefficiencies such as a 3D matrix that is continuously extending in the time dimension, only the $u(x, y, t^n)$ and $u(x, y, t^{n+1})$ values are stored. If the steady state solution is not reached the $u(x, y, t^{n+1})$ values become the $u(x, y, t^n)$ values and the solution is advanced in time again. The benefit to this method is it takes less memory to only store the solution at two times steps. It is much less resource intensive to overwrite the values in an array then to continuously modify the dimension of an array. The draw back to this method is the intermediate solution values are not stored. To access the intermediate values, one must specify a time manually and rerun a portion of the code.

Results

Grid Convergence

A grid convergence study was carried out to ensure the solution was not sensitive to grid spacing. The grid convergence was carried out for the first time, with the logic being if the solution is grid sensitive for the first-time step there was no sense in advancing the solution further in time. This was later checked against two other times, the time for the Dirichlet boundary conditions to reach steady state value and the time for the overall solution to reach a steady state value. No difference was found and the choice to carry out the grid convergence study at the first time step was justified, and that is what is present in the final version of the code. What really controls the grid convergence study is the answer to the question we have asked through the term, "*How close is close enough?*". The final close enough value used in the code was 0.001. If u was a temperature this is more than close enough for many applications.

A starting value of N=8 was used for the number of internal points in the x and y direction. The solution was advanced for a discretization with N and 2N and the results were compared. This process was repeated until the max difference between the grid was less than 0.001. The final 2N value was then used as the discretization for the "exact" solution. To compare the difference between the u_N and u_{2N} values the finer grid values were interpolated onto the coarser grid. This was done to ensure that the u values being compared were at same physical location. Consider two discretizations with grid spacings of $h_N = \frac{L}{N+1}$ and $h_{2N} = \frac{L}{2N+1}$. It is evident that the internal points from these two discretizations do not match up. Comparing values for example $u_N(h_N, y, t^n)$ and $u_{2N}(2h_{2N}, y, t^n)$ is close but is still different physical locations, and as such will have different values. Interpolating the values ensured a more accurate comparison. Rather than write a code for the interpolation Pythons scipy.interpolate.interp2d.

Once the grid convergence was finished, a portion of the code was run and the L1 and L2 norms were calculated using the "exact" solution as a reference value. The L1 and L2 norms were calculated using the steady solution. The results of this exercise were for the most part as expected and the L1 and L2 errors both decreased as the number of internal nodes increased. We can see in Figure 1 a graph of the L1 and L2 norm vs the total number of interior nodes.

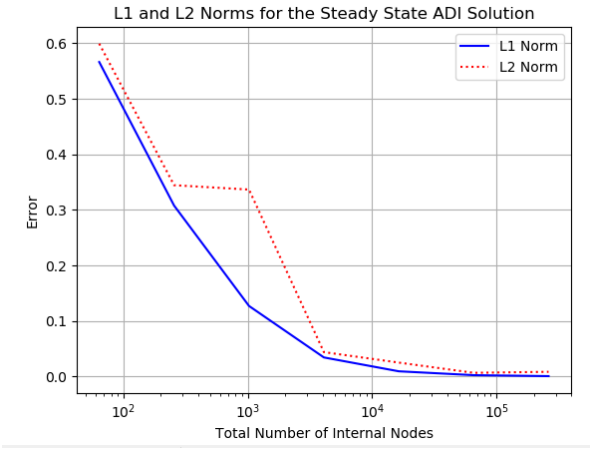

Fig. 1 L1 and L2 Norms. Both decrease as the total number of nodes increase.

The L2 norm experienced little change going from 16 internal points in each direction to 32 internal points direction. This was initially thought to have been a bug in either the ADI method or in the L2 norm function, but no error could be found. This same result was seen again for different starting values of N.

Steady State Solution

Once the grid convergence study was completed the steady state solution was then found. Using the same methodology described in the introduction, $u(x, y, t^n)$ and $u(x, y, t^{n+1})$ were deemed close enough. This method is sensitive to both the size of the time step the specified value used for determining close enough. If the time step is small and the close enough value used is too large the true steady state solution will not be found. The values used in the final version of the code and for all figures in this report of the code are $\Delta t = 0.01s$ and a close enough value of 0.005. These are of course easily changeable, and many different combinations were ran, but they eventually produce the same result and the same plot.

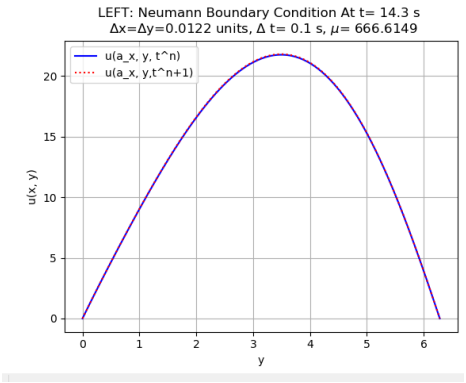

Fig. 2. Steady state solution along the Neumann Boundary condition. The parabolic shape is skewed slightly due to the larger Dirichlet condition in the top right corner of the domain.

Figure 2 shows the steady solution along the Neumann boundary condition. The steady state solution is shown in red and the previous time step is shown in blue. These lines overlap giving a clear visual indication that the steady state solution has been reached. The other boundary conditions are Dirichlet and not much is gathered from the plots so they are excluded from this report. More plots are available on Github.

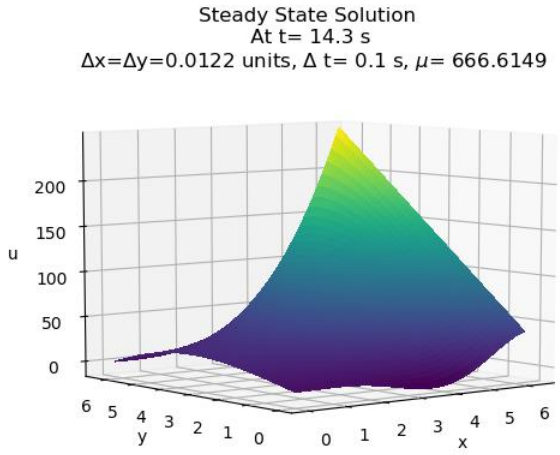

Fig. 3 Final 3D shape of the steady state solution. The Neumann boundary condition achieves a skewed parabolic shape.

Figures 3 is a 3D plot of the solution which is helpful to visual the overall shape of the solution and especially the scale of the boundary conditions. We can see the largest values occurs in the upper right of the domain. This helps to understand the behavior of the solution prior to steady state. Consider an intermediate time where the Dirichlet boundary have not yet reached there intermediate. They all scale by the same factor so the top of the domain still has the largest value. We can examine the left side of the domain where the Neumann boundary condition is enforced to see an interesting point in the solution. In Figure 4 We see the final parabolic shape is still forming and is skewed towards the top due to the larger Dirichlet condition. We also see the later time step dips below on towards the bottom of the domain due to the nature of the bottom Dirichlet condition, which does in fact reach a negative value.

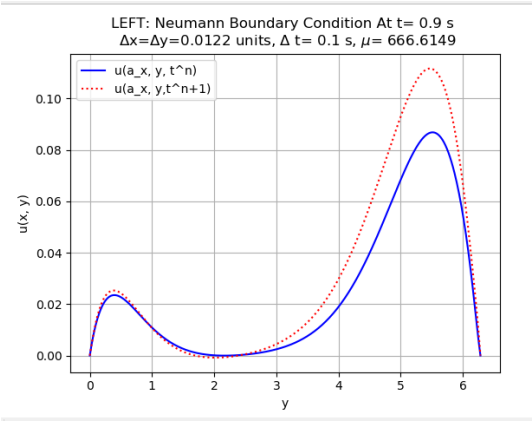

Fig. 4. Solution at x=a_x where the Neumann boundary condition is applied. The parabolic shape is forming faster towards the top of the domain due to the larger Dirichlet boundary condition on that side.

Method of Manufactured Solutions

To test the accuracy of the ADI scheme in the code the method of manufactured solution was employed. The same equation is solved except now we have a manufactured solution, whose partial derivatives act as a forcing function to the diffusion equation. A function was created in order to have similar behavior of my initial boundary conditions, Dirichlet conditions that scale with time and Neuman conditions on the left side of the domain. A few different functions were tried but they all shared the same general form.

$$u(x,y,t) = (1 - e^{-\varphi t})(\cos(x - a_x)\sin(y) + Cy$$

The function was changed a few just to run more interesting results but always a Neumann boundary condition was employed at $x = a_x$. This method indeed worked and what I initially thought was a numerical diffusion effect was in fact a simple coding error when apply the ADI scheme with a forcing function. For the ADI scheme the forcing functions in both equations used must be the average at the nth and nth+1 time steps. We can see in Figure 5 a 3D plot of the manufactured solution and in Figure 6 a zoomed in plot of the exact manufactured solution and approximate manufactured solution on the along the left side of the domain.

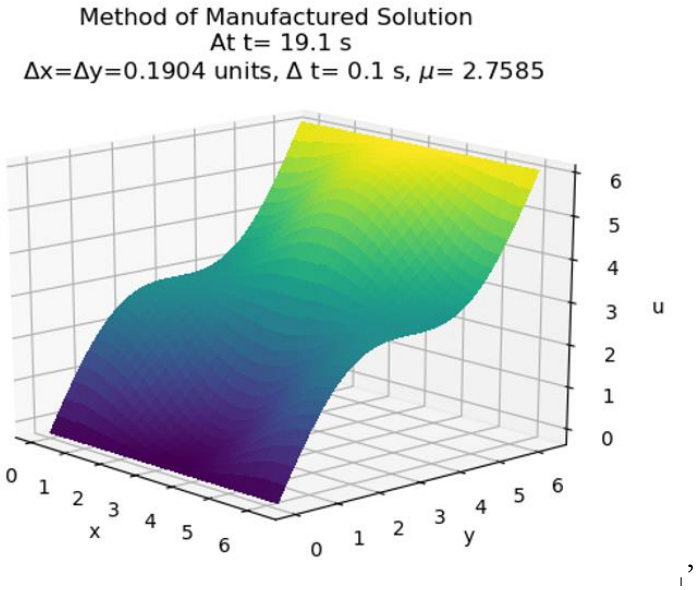

Fig. 5. One of the manufactured solutions run. Many interesting functions can be using this code.

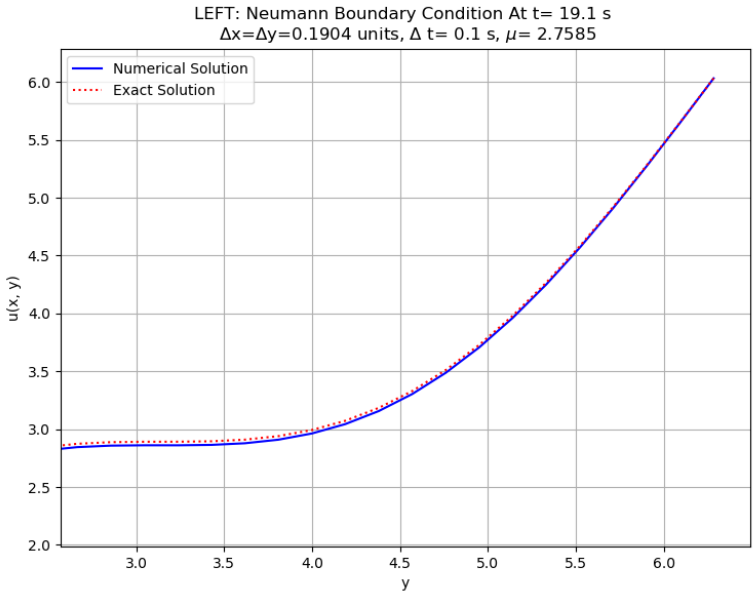

Fig. 6 Slightly zoomed in plot of the manufactured solution where the Neumann Boundary Condtion is applied.

Conclusion

The ADI method is a powerful tool for solving the diffusion equation. The ability to employ a tridiagonal matrix algorithm is highly beneficial and the banded pentadiagonal matrix is avoided. The method of manufactured solutions is also a simple yet power tool for testing the accuracy of a code. This was a very nice project and I personally enjoyed it. With more time I would have like to refine some things in the code, and focus more on the performance evaluation aspect of it. Overall the code is general, adaptable, and easily changeable with minimal effort. Some choices I made initially I was able to reverse as needed and the code was modified as the term progressed. A 2D diffusion equation was solved, a grid convergence study carried out, and the accuracy of the code was tested with the method of manufactured solutions.

Appendix A: Python Code

```
# -*- coding: utf-8 -*-

"""

Created on Sun Nov 22 11:01:42 2020

@author: johna

"""


# MECE 6397, SciComp, Grad Student Project


# 2-D Diffusion Problem

#Carryout integration till steady state solution is reached

#use ghost node method for Neumann boundary conditions

#https://github.com/jeander5/MECE_6397_Project

#12/12/2020. UPDATE! Update to project statement. Dirichlet Boundary conditions are functions of
time.

#Need to modify the code

#Modify BC with (1-exp(-lamda*t)) I will use phi instead of lambda


#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 math import exp as exp

from scipy import interpolate

#from mpl_toolkits import mplot3d


# ============================================================================

# Functions

# ============================================================================

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


# discretize the interval function
def DIF(L ,N):
    """discretizes an interval with N interior points"""
#Inputs are interval length number of interior points
#Returns the discretize domain and the interval spacing
#Includes endpoints
    h = L/(N + 1)
    x = np.linspace(0, L, N + 2)
    return(x, h)


#The given functions f and g
#these functions control the boundary conditions
#vector inputs for these functions must include the endpoints


def given_f(x, a):
    """returns the f(x) values """
#inputs are the x points and the constant in the expression. And now also time and phi.
    func_vals = [x*(x-a)*(x-a) for x in x]
```

```
    return func_vals


def given_g(x, a):
    """returns the f(x) values for the given function"""
#inputs are the x points and the constant in the expression
    func_vals = [cos(x)*(x-a)*(x-a) for x in x]
    return func_vals


def RIGHT(y, a_x, a_y, b_x, b_y):
    """returns the RIGHT boundary condition values for this problem"""
#inputs are the boundary points of the domain and the discretized y values
# I will just break this equation up for now
    uno = cos(b_x)*(b_x-a_x)*(b_x-a_x)
    dos = [(y-a_y)/(b_y-a_y) for y in y]
    tres = b_x*(b_x - a_x)*(b_x - a_x) - uno
    func_vals = [uno + dos*tres for dos in dos]
    return func_vals


#updated problem statement..............Must Modify the Dirichlet Boundary conditions
#I will call this function.....BC_Modifier
def BC_Modifier(t, phi):
    """returns the values that scales the Dirichlet Boundary conditions.
    This is just a value between 0 and 1 """
#inputs are the current time, not the time step mind, and the value phi, which is the constant in
the exponential
    func_val = (1 - exp(-phi*t))
    return func_val


#approximating( u(0,k,t=0) with a three point forward approximation)
#this is needed if the initial conditions are different
#to apply the ADI method, the starting end points must be known or calculated
#this ended up not being needed.
def TPFF(col1, col2, v, dx):
    """returns the values using the Three point forward scheme. """
#inputs are 2 arrays containing two of the points, the grid spacing, and the Neumann boundary
condition v=du/dx
```

```
    N = len(col1)
    col0 = np.ones(N)
    for k in range(0, N):
        col0[k] = -3*col1[k] + 4*col2[k] + v*2*dx
#         col0[k]=1*k+k*k/N+1.75
    return col0


#error functions
def L1_norm(u, u_ref):
    """returns the values using the L1 Norm. """
#inputs are arrays or matrices being compared
    G = abs((u - u_ref)/u)
    N = len(u)
    L1_error = (1/(N*N))*np.sum(G)
    return L1_error


def L2_norm(u, u_ref):
    """returns the values using the Three point forward scheme. """
#inputs are arrays or matrices being compared
##I brought the 1/N^2 out of the summation, all my equations assume dx=dy.
    G = abs((u - u_ref)/u)
    N = len(u)
    ##I brought the 1/N^2 out of the sqrt, all my equations assume dx=dy.
    L2_error = (1/N)*math.sqrt(np.sum(G*G))
    return L2_error



def Set_up(N, dt):
    """returns all the local variables that are needed by the ADI scheme for this problem. """
#the set up function, returns the needed local variables, discretizes, sets up Blank solution
matrix, and boundary conditions
#defines the scheme constanns for the ADI
#everything  assume the same dx and dy so i will eliminate some duplicate things that were in
previous version
#this would just need to be modified a bit for different problems
    x, dx = DIF(b_x-a_x, N)
```

```
#    x=y and dx=dy
#TP, Total points defined here so they wont need to be calculated else where
    TP = N + 2
#Sol is solution matrix for the nth time step, will be continuosly updated
    Sol = np.ones((TP, TP))
##Applying Initial condition for internal points
    Sol[1:-1, 1:-1] = Uo
#Storing Initial "unmodified' Boundary Conditions, subscript um for un modified
#Left, is the neumann
#left initial using three point forward method
    Sol[1:-1, 0] = TPFF(Sol[1:-1, 1], Sol[1:-1, 2], v, dx)
#Right
    right_um = RIGHT(x, a_x, a_y, b_x, b_y)
##Bottom
    bottom_um = given_g(x, a_x)
##Top
    top_um = given_f(x, a_x)
#    D = 1
    mu = dt/(dx*dx)
#tridiagonal constants from the ADI scheme
    a = 1 + mu
    b = -mu/2
    c = -mu/2
    d = 1 - mu
    return (Sol, x, dx, right_um, bottom_um, top_um, mu, a, b, c, d, TP)


def ADI(Sol, x, dx, right_um, bottom_um, top_um, mu, a, b, c, d, TP, q):
    """Performs The ADI Method, returns u values at the next time step"""
#This isnt so general right now. It is kinda specific to my problem. Neumann on the left side of
domain.
#Dirichlet that scale with time but eventually reach steady state


# ============================================================================
# ADI-Method
# ============================================================================
#Defining Half value matrix, for u(t=n+1/2), and Sol_next for u(t=n+1)
```

```
    N = TP - 2

    HVM = np.ones((TP, TP))

    Sol_next = np.ones((TP, TP))

#Now applying boundary conditions to the nth, nth+1/2, and nth plus 1 time step.

#I feel like this should be done outside the function, but since I need to do it for the half
values I will

#do it for the others here as as well.

    mod1 = BC_Modifier(t, phi)

    mod2 = BC_Modifier(t + dt/2, phi)

    mod3 = BC_Modifier(t + dt, phi)

    #Left, is the neumann

    #Right

    Sol[:, -1] = [x*mod1 for x in right_um]

    HVM[:, -1] = [x*mod2 for x in right_um]

    Sol_next[:, -1] = [x*mod3 for x in right_um]

    ###Bottom

    Sol[0, :] = [x*mod1 for x in bottom_um]

    HVM[0, :] = [x*mod2 for x in bottom_um]

    Sol_next[0, :] = [x*mod3 for x in bottom_um]

    ###Top

    Sol[-1, :] = [x*mod1 for x in top_um]

    HVM[-1, :] = [x*mod2 for x in top_um]

    Sol_next[-1, :] = [x*mod3 for x in top_um]


#this is the ghost node term that appears in some equations. It is calculate here

#so it doesnt have to be repeatedly calculated. For my problem its just zero.

    GNT = b*2*dx*v

# ================================================================================

# #Step "A":The t=n to t=n+ 1/2 step

# ================================================================================

    rhs_a = np.ones(N + 1)

    #rhs for step a is (N+1) because of the ghost node

    #Pre thomas algorithm set up

    av = [a]*(N + 1)

    bv = [b]*(N + 1)

    cv = [c]*(N + 1)
```

```
#different first input because of ghost node
    cv[0] = b + c


    for k in range(1, N + 1):
##first eq different
        rhs_a[0] = -b*Sol[k - 1, 0] + d*Sol[k, 0] - c*Sol[k + 1, 0] + GNT + q[k, 0]
##middle eqs
        for j in range(1, N):
            rhs_a[j] = -b*Sol[k - 1, j] + d*Sol[k, j] - c*Sol[k + 1, j] +q[k, j]
##last eq different
        rhs_a[-1] = -b*Sol[k - 1, -2] + d*Sol[k, -2] - c*Sol[k + 1, -2] - c*HVM[k,-1] + q[k, -2]
        HVM[k, 0:-1] = thomas_alg_func(av, bv, cv, rhs_a)
    # ============================================================================
    # #Step "B" the t=n+1/2 to t=n+1
    # ============================================================================
    #must solve one column at a time to preserve tridiagonal structure
    #otherwise
#    I would have to modify every Nth element of the b and c vectors, is fine for now
    #okay rhs b still is N long, I just need to do the steps N+1 one times
    rhs_b = np.ones(N)
    av_b = [a]*(N)
    bv_b = [b]*(N)
    cv_b = [c]*(N)
    #first all the u(0,k) for k=1,2...N)
    #these are a little different because of the ghost node.
    #first equation different
    rhs_b[0] = -b*HVM[1, 1]- GNT + d*HVM[1, 0] - c*HVM[1, 1] -b*Sol_next[0, 0]  + q[1, 0]
    #middle eqs
    for k in range(2, N):
        rhs_b[k-1] = -b*HVM[k, 1] - GNT + d*HVM[k, 0] - c*HVM[k, 1] + q[k, 0]
    #last equation different
        rhs_b[-1] = -b*HVM[N, 1] - GNT + d*HVM[N, 0] - c*HVM[N, 1] - c*Sol_next[N + 1, 0] + q[N,
0]
    Sol_next[1:-1, 0] = thomas_alg_func(av_b, bv_b, cv_b, rhs_b)


    #now for the next ones, u(j,k) for for j=1,2...N and k=1,2...N)
```

```
    for j in range(1, N + 1):

    #first equation different

        rhs_b[0] = -b*HVM[1, j - 1] + d*HVM[1, j] - c*HVM[1, j + 1] -b*Sol_next[0, j] + q[1, j]

    #middle eqs

        for k in range(2, N):

            rhs_b[k-1] = -b*HVM[k, j-1] + d*HVM[k, j] - c*HVM[k, j + 1] + q[k, j]

    #last equation different

        rhs_b[-1] = -b*HVM[N, j - 1] + d*HVM[N, j] - c*HVM[N, j + 1] - c*Sol_next[N + 1, j] +
q[N, j]

        Sol_next[1:-1, j] = thomas_alg_func(av_b, bv_b, cv_b, rhs_b)

    return (Sol_next)



def Manufactured_Solution(x, y, t, phi, a_x):

    """This function exact solution for the method of manufactured solutions..."""

#I am framing this manufactured solution to be similar to my boundary conditions, for example it
contains the same

# term (1-exp(-phi*t)), and du/dx at x=a_x is also zero

#the variable im using is big Q,

#Q(x,y,t)=(1-exp(-phi*t)*(cos(x-ax)*sin(y)+y))

#inputs are the arrays x, y, a single t value and the constant phi and the constant a_x

#I modified this so some of the BCS arent always zero. Always zero is no fun

    TP = len(x)

    func_vals = np.zeros((TP, TP))

    uno = (1 - exp(-phi*t))

    for k in range (TP):

        row = [uno*(cos(x-a_x)*sin(y[k])+(y[k]-0.25)) for x in x]

        func_vals[k] = row

    return func_vals


#manufactured solution partial terms

def MSPT(x, y, t, phi, a_x):

    """This function returns partial derivative terms for the the method of manufactured
solutions..."""

#I am framing this manufactured solution to be similar to my boundary conditions, for example it
contains the same

#inputs are the arrays x, y, a single t value and the constant phi
```

```
#these terms act a forcing function to the original diffusion equation
    TP = len(x)
    func_vals = np.zeros((TP, TP))
    partial_t = np.zeros(TP)
    partial_x2 = np.zeros(TP)
    fee_tee = -phi*t
    uno = (1 - exp(fee_tee))
    for k in range (TP):
        partial_t[:] = [phi*exp(fee_tee)*(cos(x-a_x)*sin(y[k])+(y[k]-math.pi)*5) for x in x]
        partial_x2[:] = [-uno*cos(x-a_x)*sin(y[k]) for x in x]
#        partial y2 is just the same as partial x2
        row = partial_t - partial_x2 - partial_x2
        func_vals[k] = row
    return func_vals


# ======================================================================
# Main program here
# ======================================================================
# ======================================================================
# Global variables
# ======================================================================
#Domain of interest
#a_x<x<b_x and a_y<y<b_y
#note not greater than or equal to
a_x = 0
a_y = 0
b_x = 2*math.pi
b_y = 2*math.pi
#Boundary Conditions, are now also functions of time
#TOP: u(x,y=by,t)=f_a(x)
#BOTTOM: u(x,y=ay,t)=ga(x)
#LEFT: NEUMANN: dudx(x=ax)=0
#defining v here to be consistent and changeable
v = 0
#phi, used in the Dirichlet Boundary Conditions
```

```
phi = 0.4999

#Initial conditions, are just zero for all points, INSIDE the boundary

#U(x,y,t)=0

#I will still define this.

Uo = 0

#defining delta t right here

dt = 0.1

# ================================================================================

# Grid Convergence. Carreid out at the first time step.

# ================================================================================

CLOSE_ENOUGH_GCS = 1*10**-3

max_diff = 2

#we will start with 8 internal points

N = 8

#starting at t=0

t =  0

print('START GCS')

while max_diff>CLOSE_ENOUGH_GCS:

#for N

    Sol_N, x, dx, right_um, bottom_um, top_um, mu, a, b, c, d, TP = Set_up(N, dt)

#    this varibale q is for a prescirbed function anad will later overritten for the method of
manual solutions.

#    it is out side the set up function for now.

#    the input would be (dt*1/2(q(x,y,t_n)+q(x,y,t_n+1))

    q=np.zeros((TP,TP))

#    q=np.ones((TP,TP))

    Sol_N_next = ADI(Sol_N, x, dx, right_um, bottom_um, top_um, mu, a, b, c, d, TP, q)

#for 2N

    N = N + N

    Sol_2N, x_ex, dx_ex, right_um_ex, bottom_um_ex, top_um_ex, mu_ex, a_ex, b_ex, c_ex, d_ex,
TP_ex = Set_up(N, dt)

    q=np.zeros((TP_ex,TP_ex))

#    q=np.ones((TP_ex,TP_ex))

    Sol_2N_next = ADI(Sol_2N, x_ex, dx_ex, right_um_ex, bottom_um_ex, top_um_ex, mu_ex, a_ex,
b_ex, c_ex, d_ex, TP_ex, q)

# Interpolate the values from the 2N grid so the points match up to the same physical location

#Interpolate the finer grid values to the coarser grid
```

```
    the_interloper = interpolate.interp2d(x_ex, x_ex, Sol_2N_next, kind='cubic')
    Sol_int = the_interloper(x, x)
#find the biggest difference between the solutions
    Diff_Matrix=abs(Sol_N_next[1:-1, 1:-1] - Sol_int[1:-1, 1:-1])
    max_diff=Diff_Matrix.max()



#Doubling the number of grid points resulted in less than "CLOSE ENOUGH' maximum difference
between the two solutions
#We will call this convereged and go back to the previous N value for the steady state solution
#The 2N grid will serve as our "Exact'" Solution
#So that the steady state solution will have N grid points, and the "Exact" solution will have 2N
grid points.
#N-final is just for reference,
N_final=round(N/2)
N_ex=N


# ================================================================================
# Advancing solution forward in time
# ================================================================================
#resetting time
t = 0
#I actually dont need to call the setup functions again, they are ready to go


#we will use the L infity  error with the u^tn+1 as the reference values
Linf_error = 2
CLOSE_ENOUGH_SS = 1*10**-3
print('START SS')
while Linf_error > CLOSE_ENOUGH_SS:
#for m in range(round(3/0.1)):
    Sol_N_next = ADI(Sol_N, x, dx, right_um, bottom_um, top_um, mu, a, b, c, d, TP, q)
    Sol_2N_next = ADI(Sol_2N, x_ex, dx_ex, right_um_ex, bottom_um_ex, top_um_ex,
                      mu_ex, a_ex, b_ex, c_ex, d_ex, TP_ex, q)
    G1 = abs(Sol_N_next[1:-1, 1:-1] - Sol_N[1:-1, 1:-1])
    Linf_error = G1.max()
    Sol_N = Sol_N_next
```

```
    Sol_2N = Sol_2N_next
#advance time
    t = t + dt


# ================================================================================
# Calculating error
# ================================================================================
#interpolating the finer grid solution to the coarser mesh
the_interloper = interpolate.interp2d(x_ex, x_ex, Sol_2N, kind='cubic')
Sol_int = the_interloper(x, x)
L1_error = L1_norm(Sol_N[1:-1, 1:-1], Sol_int[1:-1, 1:-1])
L2_error = L2_norm(Sol_N[1:-1, 1:-1], Sol_int[1:-1, 1:-1])


# ================================================================================
# Method of Manufactured solution. Uncomment and run as needed
# ================================================================================


#The way I picked my manufactured solution it sill has a Neumann boundary at x=ax, and the top
left and right
#are scalled by the same term that the updated problem is,
#So all I need to do is feed in the new un-modified BC's to my ADI function,
#and also the partial terms which are now the forcing term/ prescribed function
#getting the unmodified ms solution bondary conditions.  So just feed in a large time value
Sol_ms, x, dx, right_um, bottom_um, top_um, mu, a, b, c, d, TP = Set_up(32, dt)
right_um_ms = Manufactured_Solution(x, x, 1000, phi, a_x)[:, -1]
##the bottom and top are just zero, but I will keep it around to be general
bottom_um_ms = Manufactured_Solution(x, x, 1000, phi, a_x)[0, :]
top_um_ms = Manufactured_Solution(x, x, 1000, phi, a_x)[-1, :]



#now I dont need to call Set Up function, again I will just use the same grid, but I will set up
some blank matrices
#Sol_ms=np.zeros((TP,TP))
#intial condistion is zero for all points, including the boundaries.
#For this, I already have the the analytical solution
# resetting time
```

```
t=0

Linf_error_ms = 2

CLOSE_ENOUGH_ms = 1*10**-3

print('START MS')

while Linf_error_ms > CLOSE_ENOUGH_ms:

    q=1/4*dt*(MSPT(x, x, t, phi, a_x)+MSPT(x, x, t+dt, phi, a_x))

    Sol_ms_next = ADI(Sol_ms, x, dx, right_um_ms, bottom_um_ms, top_um_ms, mu, a, b, c, d, TP, q)

    Sol_ms_ex = Manufactured_Solution(x, x, t+dt, phi, a_x)

    G1 = abs(Sol_ms_next[1:-1, 1:-1] - Sol_ms[1:-1, 1:-1])

    Linf_error_ms = G1.max()

    Sol_ms = Sol_ms_next

#advance time

    t = t + dt

    print('MS')

#And we get the exact solution back on the boundaries because those are prescribed, elsewhere,


# ================================================================================

# Plotting, uncomment and modify as needed to make images for the report

# ================================================================================

#bottom

#x, dx = DIF(b_x, N_final)

#y=x

#y_ex=x_ex


#fig5, ax5 = plt.subplots()

#plt.grid(1)

#plt.plot(x,Sol_N[0,:],'-b')

#plt.plot(x,Sol_N_next[0,:],':r')

#plt.xlabel('x')

#plt.ylabel('u(x, y)')

#ax5.legend(['u(x, a_y)','u(x, a_y)'])

#ax5.title.set_text('BOTTOM At t= %s s \n $\Delta$x=$\Delta$y=%s units, $\Delta$ t= %s s, $\mu$=
%s'

#                   %(round(t,4),round(dx,4),round(dt,4),round(mu,4)))

###top

#fig6, ax6 = plt.subplots()
```

```
#plt.grid(1)

#plt.plot(x,Sol_N[-1,:],'-b')

#plt.plot(x,Sol_N_next[-1,:],':r')

#plt.xlabel('x')

#plt.ylabel('u(x, b_y)')

#ax6.legend(['u(x, b_y, t^n)','u(x, b_y, t^n+1)'])

#ax6.title.set_text('TOP: At t= %s s \n $\Delta$x=$\Delta$y=%s units, $\Delta$ t= %s s, $\mu$=
%s'

#                   %(round(t,4), round(dx,4),round(dt,4),round(mu,4)))

#left

#fig7, ax7 = plt.subplots()

#plt.grid(1)

#plt.plot(x,Sol_ms_next[:,0],'-b')

#plt.plot(x,Sol_ms_ex[:,0],':r')

#plt.xlabel('y')

#plt.ylabel('u(x, y)')

#ax7.legend(['Numerical Solution','Exact Solution'])

#ax7.title.set_text('LEFT: Neumann Boundary Condition At t= %s s \n $\Delta$x=$\Delta$y=%s units,
$\Delta$ t= %s s, $\mu$= %s'

#                   %(round(t,4), round(dx,4),round(dt,4),round(mu,4)))

###right

#fig8, ax8 = plt.subplots()

#plt.grid(1)

#plt.plot(x,Sol_N[:,-1],'-b')

#plt.plot(x,Sol_N_next[:,-1],':r')

#plt.xlabel('y')

#plt.ylabel('u(x, y)')

#ax8.legend(['u(b_x, y, t^n)','u(=b_x, y,t^n+1)'])

#ax8.title.set_text('RIGHT: At t= %s s \n $\Delta$x=$\Delta$y=%s units, $\Delta$ t= %s s, $\mu$=
%s'

#                   %(round(t,4), round(dx,4),round(dt,4),round(mu,4)))

###

##y=x

#3D GRAPH

#from mpl_toolkits import mplot3d

#fig10 = plt.figure()

#ax10 = plt.axes(projection='3d')
```

```
#BIGX, BIGY = np.meshgrid(x, x)

#from matplotlib import cm

#surf = ax10.plot_surface(BIGX, BIGY, Sol_ms_next, cmap=cm.viridis,

#                         linewidth=0, antialiased=False)

##fig10.colorbar(surf, shrink=0.75, aspect=5)

#ax10.set_title('u(x,y,t=%s s) \n $\Delta$x=$\Delta$y=%s units, $\Delta$ t= %s s, $\mu$= %s
'%(round(t,4),round(dx,4),round(dt,4),round(mu,4)))

#ax10.set_xlabel('x')

#ax10.set_ylabel('y')

#ax10.set_zlabel('u')

##ax10.title.set_text('Before Dirichlet Boundary Conditions are Steady \n At t= %s s \n
$\Delta$x=$\Delta$y=%s units, $\Delta$ t= %s s, $\mu$= %s'

##                    %(round(t,4), round(dx,4),round(dt,4),round(mu,4)))

##ax10.title.set_text('Steady State Solution \n At t= %s s \n $\Delta$x=$\Delta$y=%s units,
$\Delta$ t= %s s, $\mu$= %s'

##                    %(round(t,4), round(dx,4),round(dt,4),round(mu,4)))

#ax10.title.set_text('Method of Manufactured Solution \n At t= %s s \n $\Delta$x=$\Delta$y=%s
units, $\Delta$ t= %s s, $\mu$= %s'

#                    %(round(t,4), round(dx,4),round(dt,4),round(mu,4)))
```