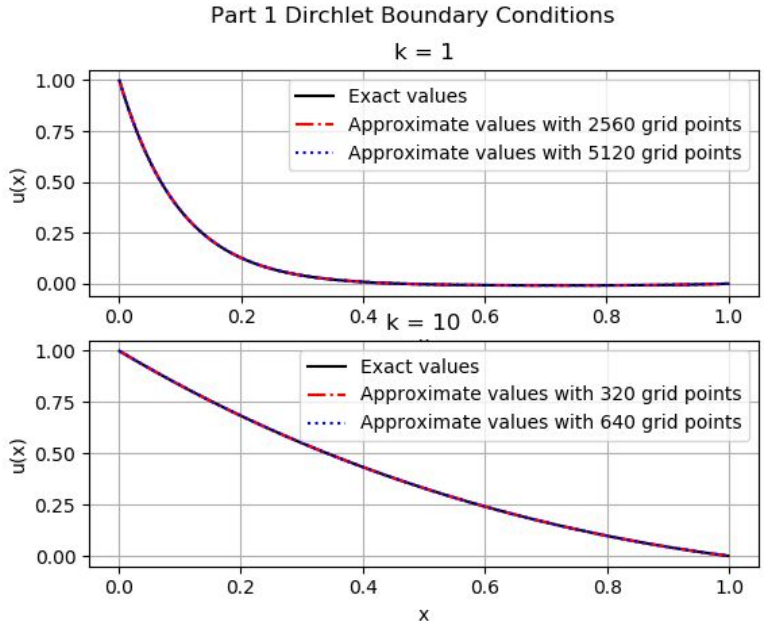
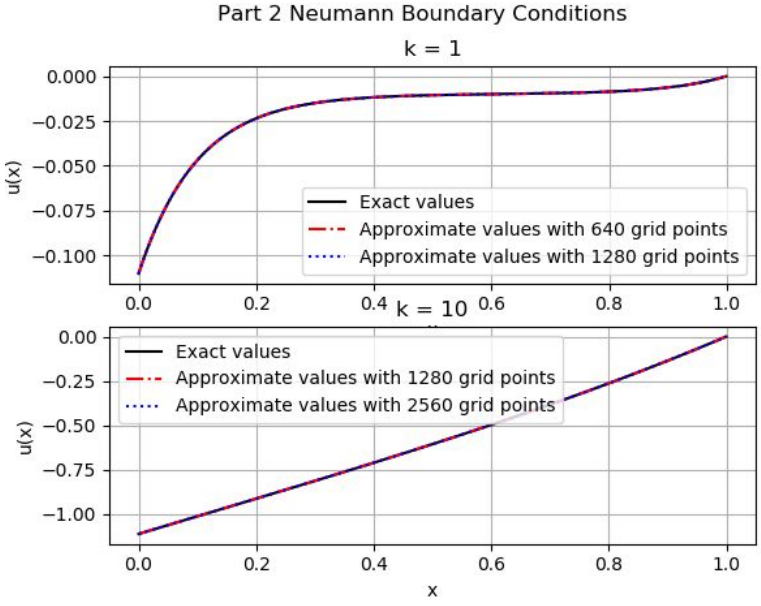
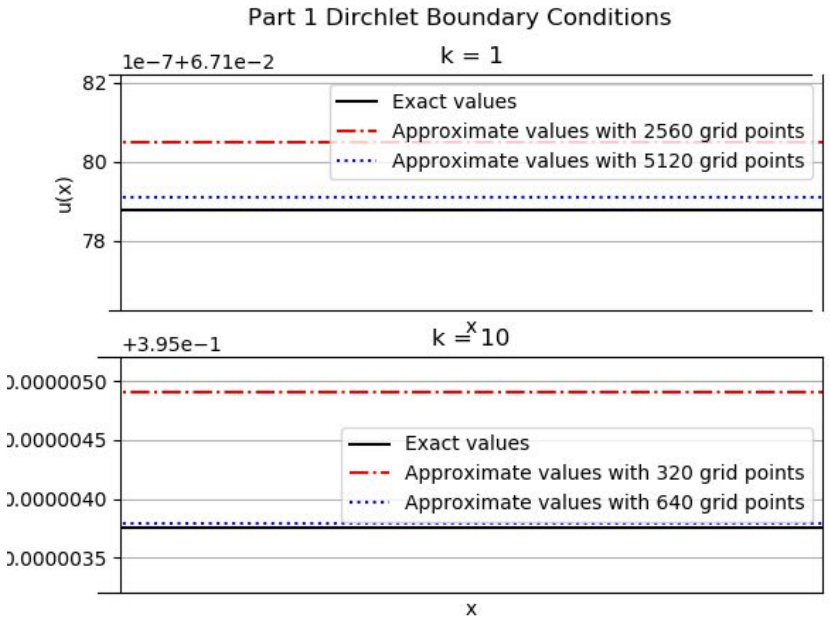
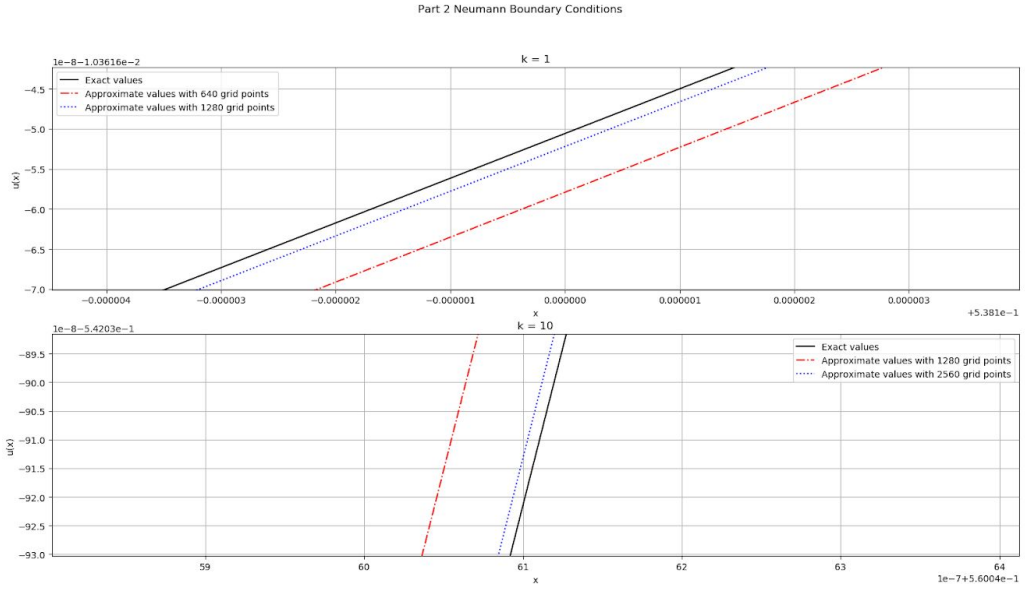
Plots-all plots are available on github.






Part 1 Dirchlet Boundary Conditions
k = 1
1e−7+6.71e−2
82
80
78
u(x)
Exact values
Approximate values with 2560 grid points
Approximate values with 5120 grid points
k = 10
+3.95e−1
0.0000050
0.0000045
0.0000040
0.0000035
Exact values
Approximate values with 320 grid points
Approximate values with 640 grid points
x


Part 2 Neumann Boundary Conditions
k = 1
1e−8−1.03616e−2
Exact values
Approximate values with 640 grid points
Approximate values with 1280 grid points
u(x)
x
+5.381e−1
k = 10
1e−8−5.4203e−1
Exact values
Approximate values with 1280 grid points
Approximate values with 2560 grid points
1e−7+5.6004e−1

Some Table Values-More table values are available on github in an excel file.

Dirchlet, k=1

N=10					
x	0.0909091	0.181818	0.272727	0.363636	
u_appx	0.8467099746	0.7086820151	0.5847753946	0.4739660914	
u_exact	0.8466702864	0.7086128245	0.5846858575	0.4738644899	
N=20					
x	0.047619	0.0952381	0.142857	0.190476	
u_appx	0.9177110594	0.83977067	0.7660020962	0.6962380623	
u_exact	0.9177049832	0.8397593255	0.765986246	0.6962184273	
N=320					
x	0.00311526	0.00623053	.....	.....	..
u_appx	0.9944796278	0.9889786118	.....	.....	..
u_exact	0.994479626	0.9889786082	.....	.....	..
N=640					
x	0.00156006	0.00312012	0.00468019	0.00624025	..
u_appx	0.9972330843	0.9944710295	0.9917138288	0.9889614755	..
u_exact	0.9972330841	0.994471029	0.9917138281	0.9889614746	..

Dirchlet, k=10

N=10				
x	0.0909091	0.181818	0.272727	
u_appx	0.4087717783	0.1636359354	0.06200086556	
u_exact	0.3969201644	0.153946528	0.05605825804	
N=20				
x	0.047619	0.0952381	0.142857	
u_appx	0.6201160523	0.3831155633	0.2352569253	
u_exact	0.617357056	0.3796805169	0.2320493242	
N=2560				
x	0.000390472	0.000780945	.....	.
u_appx	0.9960639237	0.9921431868	.....	.
u_exact	0.9960639212	0.9921431818	.....	.
N=5120				
x	0.000195274	0.000390549	0.000585823	
u_appx	0.9980296555	0.9960631547	0.9941004903	
u_exact	0.9980296551	0.9960631541	0.9941004894	

Neumann, k=1

N=10					
x	0	0.0909091	0.181818	0.272727	
u_appx	-1.11244015	-1.021995688	-0.9317330079	-0.8409061382	
u_exact	-1.113539882	-1.022975017	-0.9326001582	-0.8416678925	
N=20					
x	0	0.047619	0.0952381	0.142857	
u_appx	-1.11323782	-1.06574716	-1.018405587	-0.9711057493	
u_exact	-1.113539882	-1.06603159	-1.018673058	-0.9713568766	
N=1280					
x	0	0.00078064	.....	.....	..
u_appx	-1.113539801	-1.112759196	.....	.....	..
u_exact	-1.113539882	-1.112759277	.....	.....	..
N=2560					
x	0	0.000390472	0.000780945	0.00117142	..
u_appx	-1.113539862	-1.113149398	-1.112758952	-1.112368523	..
u_exact	-1.113539882	-1.113149418	-1.112758972	-1.112368543	..

Neumann, k=10

N=10				
x	0	0.0909091	0.181818	
u_appx	-0.1010354019	-0.0477442456	-0.02564668076	-(
u_exact	-0.1099990916	-0.05028772179	-0.02622918919	-(
N=20				
x	0	0.047619	0.0952381	
u_appx	-0.1072796253	-0.0706900137	-0.04786230993	-(
u_exact	-0.1099990916	-0.0721135024	-0.0485807782	-(
N=640				
x	0	0.00156006	.....	.
u_appx	-0.1099960494	-0.1084481555	.....	.
u_exact	-0.1099990916	-0.108451135	.....	.
N=1280				
x	0	0.00078064	0.00156128	
u_appx	-0.1099983298	-0.1092207366	-0.10844919	
u_exact	-0.1099990916	-0.1092214905	-0.108449936	

Ipython Console Output

runfile('C:/Users/johna/Desktop/FALL 2020/SCI COMP/HW4 Comp/Anderson_John_HW_4_Computational.py', wdir='C:/Users/johna/Desktop/FALL 2020/SCI COMP/HW4 Comp')

For k = 1

Part 1. Dirchlet Boundary Conditions

320 grid points needed
Doubling the grid points would result in less than 0.001 difference between u values for the closest grid points

Log2(Error_N/Error_2N) = 2.0

Part 2. Neumann Boundary Conditions

1280 grid points needed
Doubling the grid points would result in less than 0.001 difference between u values for the closest grid points

Log2(Error_N/Error_2N) = 2.02

For k = 10

Part 1. Dirchlet Boundary Conditions

2560 grid points needed
Doubling the grid points would result in less than 1e-05 difference between u values for the closest grid points

Log2(Error_N/Error_2N) = 2.0

Part 2. Neumann Boundary Conditions

640 grid points needed
Doubling the grid points would result in less than 1e-05 difference between u values for the closest grid points

Log2(Error_N/Error_2N) = 2.0

Python Code

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct  9 14:07:44 2020
@author: johna
"""
#https://github.com/jeander5/MECE_6397_HW4_Computational
# MECE 6397, SciComp, Problem 4, Computational
#Solve the Helmholtz's equation for 1. Dirchlet. 2. Neumann

#imports
from math import sinh as sinh
from math import cosh as cosh
from math import log2 as log2
import numpy as np
import matplotlib.pyplot as plt

#Constants given in problem statement, constant for both boundary conditions
#Interval length, u(x=0) for Dirchlet, v is constant for the Neumann, A is exact solution of f(x)
L = 1
U_0 = 1
v = 1
A = 1
K = [1, 10]

#The N_INITIAL value must be greater than 2
#N is defined in the for loop and changes during the grid convergence study
N_INITIAL = 10

#Helmholtz Thomas Algorithm Function, for Dirchlet part 1
def thomas_alg_one(N, h, lamda, U_0, A):
    """returns exact u values for the function from Part 1"""
#inputs are N, lamda, U_0=u(x=0), and for this problem A.
#Pre Thomas Algorithm set up. For this problem these values are all constant
    a = -(2-lamda*h**2)
    b = 1
    c = 1
#Right hand side, the f*h**2 from the pseudocode
    rhs = A*h**2
```

```
    alpha = [0]*N
    g = [0]*N
    u_appx = [0]*N
#Following the pseudocode
#Zeroth element of this list corresponds to the first subscript in Thomas Algorithm
    alpha[0] = a
    g[0] = rhs-U_0
    for j in range(1, N):
        alpha[j] = a-(b/alpha[j-1])*c
        g[j] = rhs-(b/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*u_appx[-j])/alpha[-1-j]
    return u_appx

#Helmholtz Thomas Algorithm Function, for Neumann part 2
def thomas_alg_two(N, h, lamda, v, A):
    """returns approximate u values for the function from Part 2"""
#Pre Thomas Algorithm set up.
#I now need to make N one point larger to incorporate the ghost node method for
#u(x=0) which is unknown. But I am still keeping h the same
    N = N+1
#These values are constant but the c's are not
    a = -(2-lamda*h**2)
    b = 1
#I now need c to be a list because they are now not all the same
#Or I could use some conditional statements but I want to still follow the
#Pseudocode for the algorithm closely
    c = [1]*N
    c[0] = 2
#This line is added because of the ghost node method
#rhs is right hand side
    rhs = A*h**2
    alpha = [0]*N
    g = [0]*N
    u_appx = [0]*N
#Following the pseudocode
#Zeroth element of this list does in fact correspond to  subscript zero in Thomas Algorithm
#Because of the ghost node method
    alpha[0] = a
```

```
    g[0] = rhs+2*h*v
    for j in range(1, N):
        alpha[j] = a-(b/alpha[j-1])*c[j-1]
        g[j] = rhs-(b/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, for the Helmhotlz Dirchlet part 1 problem
def u_exact_func(k, L, x, A, U_0):
    """returns exact u values for the function from Part1"""
    func_vals = [((sinh(k*(L-x))+sinh(k*x))/sinh(k*L)-1)*A/k**2+
                U_0*sinh(k*(L-x))/sinh(k*L) for x in x[1:-1]]
#x[1:-1] I dont need u(x=0) or u(x=L) because they are given
    return func_vals

#u exact function, for the Helmhotlz Neumann Part 2 problem
def u_exact_func2(k, L, x, A, v):
    """returns exact u values for the function from Part 2"""
    func_vals = [((cosh(k*x)/cosh(k*L))-1)*(A/k**2)
                -(v/k)*(sinh(k*(L-x))/cosh(k*L)) for x in x[0:-1]]
#x[0:-1]  I dont need  u(x=L) because it is given
    return func_vals

#Pre-loop plot formatting
#Turning on grid, and setting up subplots, titles, and labels
#Legends are defined inside the for loop
plt.rcParams['axes.grid'] = True
fig1, (ax1, ax2) = plt.subplots(2)
fig1.suptitle('Part 1 Dirchlet Boundary Conditions')
fig2, (ax3, ax4) = plt.subplots(2)
fig2.suptitle('Part 2 Neumann Boundary Conditions')
ax1.title.set_text('k = %s'%(K[0]))
ax1.set_xlabel('x')
ax1.set_ylabel('u(x)')
ax2.title.set_text('k = %s'%(K[1]))
ax2.set_xlabel('x')
ax2.set_ylabel('u(x)')
ax3.title.set_text('k = %s'%(K[0]))
```

```
ax3.set_xlabel('x')
ax3.set_ylabel('u(x)')
ax4.title.set_text('k = %s'%(K[1]))
ax4.set_xlabel('x')
ax4.set_ylabel('u(x)')

#Changing this number right here changes my results!
#Note! Very Important!
#Difference between N and 2N approximation
CLOSE_ENOUGH = 1*10**-3
#This is really the number the controls the grid convergence study
#As in "how close is close enough?"

#Outer for loop for the different k values
LEN_K = len(K)
for n in range(LEN_K):
    k = K[n]
    print('\nFor k = %s \n'%(k))

#Using a different value here for k=10
    if n == 1:
        CLOSE_ENOUGH = 1*10**-5

#Note: lamda in the Helmholtz equation is defined here
    lamda = -k**2

#Part 1, Dirchlet
    print('Part 1. Dirchlet Boundary Conditions \n')

#Grid Convergence Study

#Resetting the N value. Needs to come before both grid convergence studies
    N = N_INITIAL
#I am using the Flag so I dont have to call the function before and inside the while statement
    Flag = 0
    while Flag == 0:
#Comparing values near the middle of the interval
        check_val = round(N/2)
#Calling the functions
        h = L/(N+1)
```

```
        u_appx = thomas_alg_one(N, h, lamda, U_0, A)
        N2 = 2*N
        h2 = L/(N2+1)
        u_appx_next = thomas_alg_one(N2, h2, lamda, U_0, A)
#I still need to be comparing u values for the closest x points, hence 2*check_val+1
        if abs(u_appx[check_val]-u_appx_next[2*check_val+1]) < CLOSE_ENOUGH:
            Flag = 1
            print('%s grid points needed' %(N))
            print('Doubling the grid points would result in less than %s '
                'difference between u values for the closest grid points \n'
                %(CLOSE_ENOUGH))
        else:
            N = N+N

#ls for legend string. This is here so I dont need to store the N and N2 value.
#I am storing the string instead
        ls1 = ('Approximate values with %s grid points'%(N))
        ls2 = ('Approximate values with %s grid points'%(N2))

#Note: here are the exact value function calls. x values are also defined here
    x1 = np.linspace(0, L, N+2)
    u_exact = u_exact_func(k, L, x1, A, U_0)
    x2 = np.linspace(0, L, N2+2)
    u_exact_next = u_exact_func(k, L, x2, A, U_0)

#formal order of accuracy, fooa

#Finding max absolute error, this method proved to be faster than others
    stored_val = (u_appx[0]-u_exact[0])
    for j in range(1, N):
        next_val = u_appx[j]-u_exact[j]
        if next_val > stored_val:
            stored_val = next_val
    stored_val2 = (u_appx[0]-u_exact[0])
    for j in range(1, N*2):
        next_val2 = u_appx_next[j]-u_exact_next[j]
        if next_val2 > stored_val2:
            stored_val2 = next_val2

    fooa1 = round(log2(stored_val/stored_val2), 2)
```

```
    print('Log2(Error_N/Error_2N) = %s'%(fooa1))

#Part 2, Neumann
    print('\nPart 2. Neumann Boundary Conditions \n')

#Grid Convergence Study

    N = N_INITIAL
    Flag = 0
    while Flag == 0:
#Comparing values near the middle
        check_val = round(N/2)
#Calling the functions
        h3 = L/(N+1)
        u2_appx = thomas_alg_two(N, h3, lamda, v, A)
        N2 = 2*N
        h4 = L/(N2+1)
        u2_appx_next = thomas_alg_two(N2, h4, lamda, v, A)
# I still need to be comparing u values for the closest x points, hence 2*check_val+1
        if abs(u2_appx[check_val]-u2_appx_next[2*check_val+1]) < CLOSE_ENOUGH:
            Flag = 1
            print('%s grid points needed' %(N))
            print('Doubling the grid points would result in less than %s '
                'difference between u values for the closest grid points \n'
                %(CLOSE_ENOUGH))
        else:
            N = N+N

#Note: here are the exact value function calls. x values are also defined here
    x3 = np.linspace(0, L, N+2)
    u2_exact = u_exact_func2(k, L, x3, A, v)
    x4 = np.linspace(0, L, N2+2)
    u2_exact_next = u_exact_func2(k, L, x4, A, U_0)

#formal order of accuracy, fooa

#Finding max absolute error, this method proved to be faster than others

    stored_val3 = (u2_appx[0]-u2_exact[0])
    for j in range(1, N):
```

```
        next_val3 = u2_appx[j]-u2_exact[j]
        if next_val3 > stored_val3:
            stored_val3 = next_val3
    stored_val4 = (u2_appx_next[0]-u2_exact_next[0])
    for j in range(1, N*2):
        next_val4 = u2_appx_next[j]-u2_exact_next[j]
        if next_val4 > stored_val4:
            stored_val4 = next_val4

    fooa2 = round(log2(stored_val3/stored_val4), 2)
    print('Log2(Error_N/Error_2N) = %s'%(fooa2))

#Plotting, inside the for loop still
#k==1 plots
    if n == 1:
        ax1.plot(x2[1:-1], u_exact_next, 'k')
        ax1.plot(x1[1:-1], u_appx, '-.r')
        ax1.plot(x2[1:-1], u_appx_next, ':b')
        ax1.legend(['Exact values', ls1, ls2])
        ax3.plot(x4[0:-1], u2_exact_next, 'k')
        ax3.plot(x3[0:-1], u2_appx, '-.r')
        ax3.plot(x4[0:-1], u2_appx_next, ':b')
        ax3.legend(['Exact values', 'Approximate values with %s grid points'%(N),
                'Approximate values with %s grid points'%(N2)])
#k==10 plots
    else:
        ax2.plot(x2[1:-1], u_exact_next, 'k')
        ax2.plot(x1[1:-1], u_appx, '-.r')
        ax2.plot(x2[1:-1], u_appx_next, ':b')
        ax2.legend(['Exact values', ls1, ls2])
        ax4.plot(x4[0:-1], u2_exact_next, 'k')
        ax4.plot(x3[0:-1], u2_appx, '-.r')
        ax4.plot(x4[0:-1], u2_appx_next, ':b')
        ax4.legend(['Exact values', 'Approximate values with %s grid points'%(N),
                'Approximate values with %s grid points'%(N2)])
```