College of Engineering

Coursework Submission Sheet

Swansea University
Prifysgol Abertawe

Coursework Title: Finite Difference Method	Student number: 958840
Coursework number (i.e. CW1 CW2): CW2	Name: Deepak Singh
Module code: EG-396	Email: 958840@swansea.ac.uk
Module title: Computational Fluid Dynamics	Degree course: Aerospace Engineering
Submission deadline: 02/04/21	
Lecturer: Prof. P. Nithiarasu	

By submitting this coursework, I certify that this is all my own work.

Submission date 02/04/21

Student signature if this is a non-electronic submission……

SPLD Students

Please tick this box if you are officially recognised by the University as an SPLD student. ☐

A very good report. The code is written in python. In general the presentation is good apart from minor comments. Please check my minor comments.

Total marks: 100 %.

Well done!

958840

CFD Assignment 2

Notation

ϕ	Unknown Variable (Usually Temperature)
$\tilde{\phi}$	Approximate Unknown Variable
ϕ_A	Analytical Unknown Variable
u	Convective Velocity
k	Diffusion coefficient
t	time
x	X spatial value
l	Length of element
L	Length of complete bar/total elements
w	Weight function
N	Shape function
Q	Load variable (Usually mass flux)
$[\mathrm{K}]^{\mathrm{e}}$	Local Stiffness matrix
$\{\mathrm{u}\}^{\mathrm{e}}$	Local Displacement Vector
$\{f\}^{e}$	Local Force Vector
α	Peclet Scalar
k_b	Added Balancing-Diffusion value
Pe	Peclet Number

good idea

1 Introduction

Many physical phenomena can be explained using partial differential equations; however, an analytical solution may be impossible or currently not proved. The finite element method is a numerical approach to solve partial differential equations. The method works by modelling the spatial or time problems into finite nodes and elements and then discretising problem into a set of linear equations which can be solved. The approach to solving these partial differential equations is not singular, and there are different approaches that can be taken to discretize the equations. This short paper will investigate the standard Galerkin and the Balancing Diffusion method, comparing the stability properties identified from the different methods.

2 Governing equations and discretization

Standard Galerkin

The 1D Convection-Diffusion equation can be defined as:

$$\frac{\partial \phi}{\partial t} - k\frac{\partial^2 \phi}{\partial x^2} + u\frac{\partial \phi}{\partial x} = 0 \qquad (1)$$

For this 1D finite element problem, the time variable isn't considered as this is a steady-state problem, therefore, (1) becomes:

$$-k\frac{\partial^2 \phi}{\partial x^2} + u\frac{\partial \phi}{\partial x} = 0 \qquad (2)$$

In the finite element method, the strong form of (2) is discretised into a weak form. The strong form of (2) can be developed be known as:

$$-k\frac{d}{dx}\left(\frac{d\phi}{dx}\right)+u\frac{d\phi}{dx}=0 \tag{3}$$

The weak form can be developed by replacing the unknown variable with an approximate solution. A weight function and integral with finite limits of the spatial element domain is applied to (3):

$$-\int_0^l k\frac{d}{dx}\left(\frac{d\tilde{\phi}}{dx}\right)w\,dx+\int_0^l u\frac{d\tilde{\phi}}{dx}w\,dx=0 \tag{4}$$

For this 1D finite element problem, the same shape functions will be used to discretise the strong form. They can be defined as:

$$N_1=1-\frac{x}{l} \tag{5a}$$

$$N_2=\frac{x}{l} \tag{5b}$$

For the standard Galerkin Method, the shape functions are equal to the weight functions. Therefore:

$$w=N=\begin{bmatrix}N_1\\N_2\end{bmatrix} \tag{6}$$

Equation (4) can be developed further by applying integration by parts. The weak form can be developed as:

$$\int_0^l k\frac{d\phi}{dx}\frac{dw}{dx}dx+\int_0^l u\frac{d\tilde{\phi}}{dx}w\,dx=[-Qw]_0^l \tag{7}$$

Where:

$$k\frac{d\tilde{\phi}}{dx}=-Q \tag{8}$$

For this problem, there will no load ($Q=0$), however, the load vector can still be developed. For the convection diffusion problem, the load vector will usually contain the mass flux. Using the Galerkin method, (7) can be developed further by using (6):

$$\int_0^l k\left[\frac{dN}{dx}\right]\left[\frac{dN}{dx}\right]\{\phi\}dx+\int_0^l u\left[\frac{dN}{dx}\right][N]\{\phi\}\,dx=\left[-Q[N]\right]_0^l \tag{9}$$

By calculating the derivatives of the shape functions and calculating the integrals in (9), the stiffness matrix, displacement vector and load vector can be developed as:

$$\left(\frac{k}{l}\begin{bmatrix}1&-1\\-1&1\end{bmatrix}+\frac{u}{2}\begin{bmatrix}-1&-1\\1&1\end{bmatrix}\right)\begin{Bmatrix}\phi_1^e\\\phi_2^e\end{Bmatrix}=\frac{Q}{2}\begin{Bmatrix}-1\\-1\end{Bmatrix}=0 \tag{10a}$$

$$[\mathrm{K}]^{\mathrm{e}}\{\mathrm{u}\}^{\mathrm{e}}=\{\mathrm{f}\}^{\mathrm{e}} \tag{10b}$$

Balancing Diffusion

The balancing diffusion method can be shown to equal to the Petrov-Galerkin method. In this method, the weight functions are not equal to the shape functions. Therefore, the weight function can be defined as:

$$w=N+\alpha w^* \tag{11}$$

The value of α can be defined as:

$$\alpha = \coth|Pe| - \frac{1}{|Pe|} \tag{12}$$

The Pe value will be discussed in more detail further on. The added weight function w^* can developed as:

$$w^* = \frac{l}{2}\frac{u}{|u|}\frac{dN}{dx} \tag{13}$$

In the Balancing Diffusion procedure, (13) and (12) can be equated to a new diffusion coefficient value:

$$k_b = \frac{1}{2}\alpha u l \tag{14}$$

The standard Galerkin strong from shown in (4) can altered by using (14):

$$-\int_0^l (k + k_b)\frac{d}{dx}\left(\frac{d\tilde{\phi}}{dx}\right) w\, dx + \int_0^l u\frac{d\tilde{\phi}}{dx} w\, dx = 0 \tag{15}$$

The standard finite element method discretization procedure is repeated. The stiffness matrixes for the Balancing Diffusion matrix can be developed as:

$$\left(\frac{(k + k_b)}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} + \frac{u}{2}\begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}\right) = [\mathrm{K}]^{\mathrm{e}} \tag{16}$$

Peclet Number

The Peclet number is a dimensionless number defined as the ratio of the rate of convection by the rate of diffusion. Therefore, for a 1D convection-diffusion problem, the Pe number can developed as:

$$Pe = \frac{ul}{2k} \tag{17}$$

In the standard Galerkin method solution, as $Pe \to \infty$, the solved solution is purely oscillatory. However, in the Balanced-Diffusion method, stabilisation can be seen. As $Pe \to \infty$, there should not be an oscillatory solution and the solution should bear similar relations to the analytical method. The analytical method can be defined as:

$$\phi_A = \frac{e^{\frac{ux}{k}} - e^{\frac{uL}{k}}}{1 - e^{\frac{uL}{k}}} \tag{18}$$

3 Code Structure

Due to the post-processing requirements for this project (Plots for results and discussion), the script written for this assignment contains 5 minor functions and a singular main function. The minor functions written do the following:

- Convert local stiffness, load vectors into the global stiffness, load vectors.
- Minus 1 function (Used for indexing in dictionaries)

- Compute $\coth x$
- Compute analytical solution shown in (18)
- Compute u, depending on Pe

The singular main function contains the pre-processing and main-processing elements. During the post-processing stage, the main function is used with the minor functions to produce the required plots for the results and discussion section.

Pre-Processing

```
49 ################################## Main Function ########################
50 def Main(PeV,Type,k,L,Nodes,Q):
51
52
53 # ---------------------------------- Pre-Processing ----------------------#
54     h = L/(Nodes-1)
55     ndof = 1
56
57
58     # Boundary Conditions
59     phi = []
60     for i in range(0,Nodes):
61         phi.append("Free")
62
63     phi[0] = 1 #phi @ 0
64     phi[Nodes-1] = 0 #phi @ L
65
66 #Type == 1 (Balancing Diffusion) Type >1 && 1<(Standard Galerkin)
67     u = uscaler(PeV,k,L,Nodes) #Calculates u scaler depending on Pe number
68
69     # Balancing Diffusion k values computaion
70     if Type == 1:
71
72         if PeV == 0:
73             PeV = 1E-8
74             alpha = Coth(abs(PeV))-(1/abs(PeV)) #Divide by zero warning
75
76         else:
77             alpha = Coth(abs(PeV))-(1/abs(PeV))
78
79         kb = 0.5*alpha*u*h
80         k+= kb
81
82     # Stiffness and loadvectors
83     K = np.zeros((Nodes,Nodes))
84     F = np.zeros((Nodes))
```

Figure 1: Pre-Processing code

The boundary conditions can be seen in the main function at line 58. An initial ϕ vector is set up depending on the number of nodes inputted into the function. The unknown ϕ value is set to “free”. As the main function contains both the Standard-Galerkin method and Balancing-Diffusion processes, an if statement is used in the code to differentiate the type of method required. This can be seen in figure 1, in line 70.

Main-Processing

```
86 #---------------------------- Main Processing----------------------------#
87
88
89     for i in range(0,Nodes-1):
90         #Generate local stifness matrices
91         ke = (u/2)*np.array([[-1,1],[-1,1]])+(k/h)*np.array([[1,-1],[-1,1]])
92         fe = (Q/2)*np.array([[1],[1]])
93
94         #Local Matix ==> Global Matrix
95         IN = Matrix([0:1,1:2])
96         OUT = {0:1,1:2}
97
98         for j in range(0,len(OUT)):
99             OUT[j] = ndof*i+j
100
101        K = Gather(IN,OUT,ke,K,1)
102        F = Gather(IN,OUT,fe,F,2)
103
104    A = K
105    G = F
106    c = 0
107    phi_loc = []
108
109    # Condense matrix by enforcing Boundary Condtions and using Gauss ELim
110    for i in range(0,Nodes):
111        if phi[i] == "Free":
112            phi_loc.append(i)
113
114        else:
115            G += -phi[i]*A[:,i-c]
116            A = np.delete(A,i-c,axis=0)
117            A = np.delete(A,i-c,axis=1)
118            G = np.delete(G,i-c,axis=0)
119            c += 1
120
121    # Solve missing phi values
122    phiv = np.dot(np.linalg.inv(A),G)
123
124    # Replaec phi ca
125    c = 0
126    for i in range(0,Nodes):
127        if i == phi_loc[c]:
128            phi[i] = phiv[c]
129            c += 1
130
131            if c == len(phi_loc):
132                c = 0
133    else:
134        phi[i] = phi[i]
135
136    return phi
```

Figure 2: Main-processing code

A loop is set up at line 89 to calculate the local stiffness matrices and load vector (load vector is always equal to 0 for this project). After the matrices and vectors are set up, the “Gather” function is used to convert the local matrices and vectors to the global ones. After the global matrices and vectors have been calculated, the matrices are condensed using the boundary conditions specified in the pre-processing stage. Also, the load vector changes due to the boundary conditions applied. In line 122, a solution is found for ϕ by calculated in the inverse of the condensed stiffness matrix and multiplying it with the load vector. Finally, the ϕ vector is replaced and outputted out of the function.

Post-Processing

```
138 #-------------------------- Post-Processing--------------------------#
139 # Figure (1)
140 #Standard Galerkin discretization is oscillatory beyond an element
141 #Peclet number ofunity. Plot the scalar variable value at different
142 #Peclet numbers along the domainlength,
143
144 Pe1 = np.array((0,0.1,0.5,1,2,10,100,10000))
145 k = 1
146 L = 1
147 Nodes = 10
148 x = np.linspace(0,L,Nodes)
149
150
151 for i in range(0,len(Pe1)):
152     phi = Main(Pe1[i],0,k,L,Nodes,0)
153     plt.figure(1)
154     u = uscaler(Pe1[i],k,L,Nodes)
155     plt.plot(x,phi,'--o',label='Pe = '+str(Pe1[i])+' , u = '+str(round(u,2)))
156
157 plt.grid(True)
158 plt.ylabel(r'$\phi$')
159 plt.xlabel(r'$x$')
160 plt.legend()
161 plt.show()
```

Figure 3: Post-Processing example

An example of the post-processing structure can be seen in figure 3. The main function is used to calculate the different ϕ solutions. A loop is used here, therefore, allowing a plot is to be produced for ϕ solutions with different Peclet numbers. Further post-processing code can be seen in the appendix. The structure for these plots is like the example shown in figure 3. There are some minor differences due to the plot requirements of the report.

4 Results and Discussion

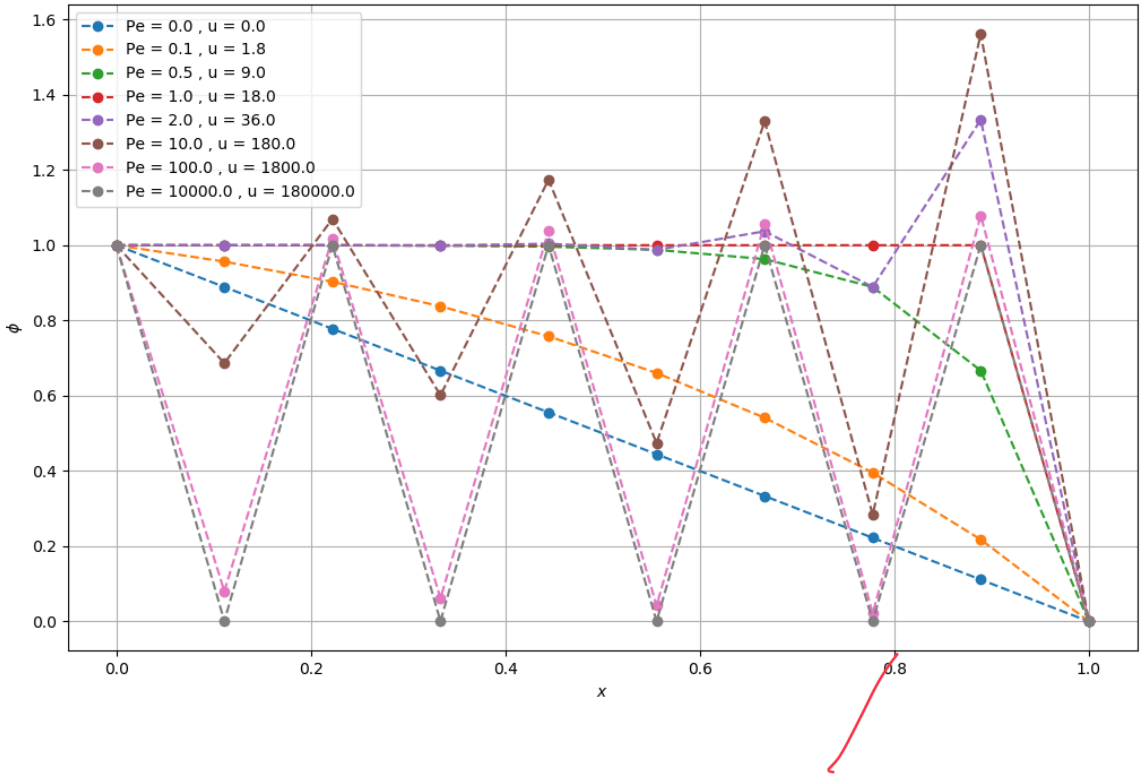

Figure 4:Convecton-Diffusion Solution using the Standard Galerkin Method

Figure 4 displays the results obtained for the FEM standard Galerkin code. The plot displays the ϕ value along the length of the bar. For this example, the input values were:

- $k = 1$
- $L = 1$
- $\text{Nodes} = 10$
- $\phi(0) = 1$ *(B.C)*
- $\phi(L) = 0$ *(B.C)*

From the ranges of $0 \leq Pe \leq 1$, the solution is stable and there are no oscillations in the plot. Due to the value of u being determined from the Pe value, the general shape of the plot changes the Pe value increases. When Pe is greater to 1, oscillations start to occur in the solution. The plot displays the stability of the oscillations getting worse, the greater the Pe value. This can be further displayed in figure 2, which has solutions for ϕ ranging from $1 < Pe \leq 10000$. In figure 5, the nodal values are increased to 50, allowing clearer oscillations to be plotted. When the Peclet number is equal to 0, the solution is linear. This linear solution only occurs at when $Pe = 0$, for the stable Pe values.

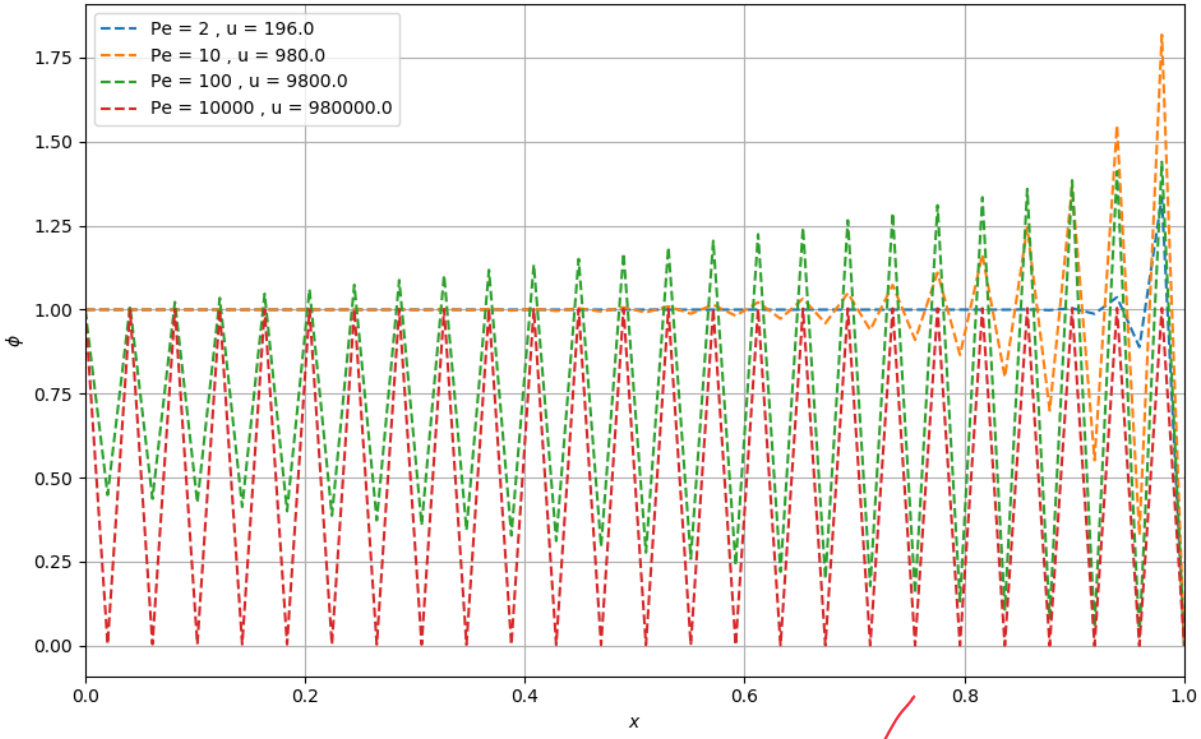

Figure 5: Plot displaying the oscillations with varying Peclet numbers greater than 1.

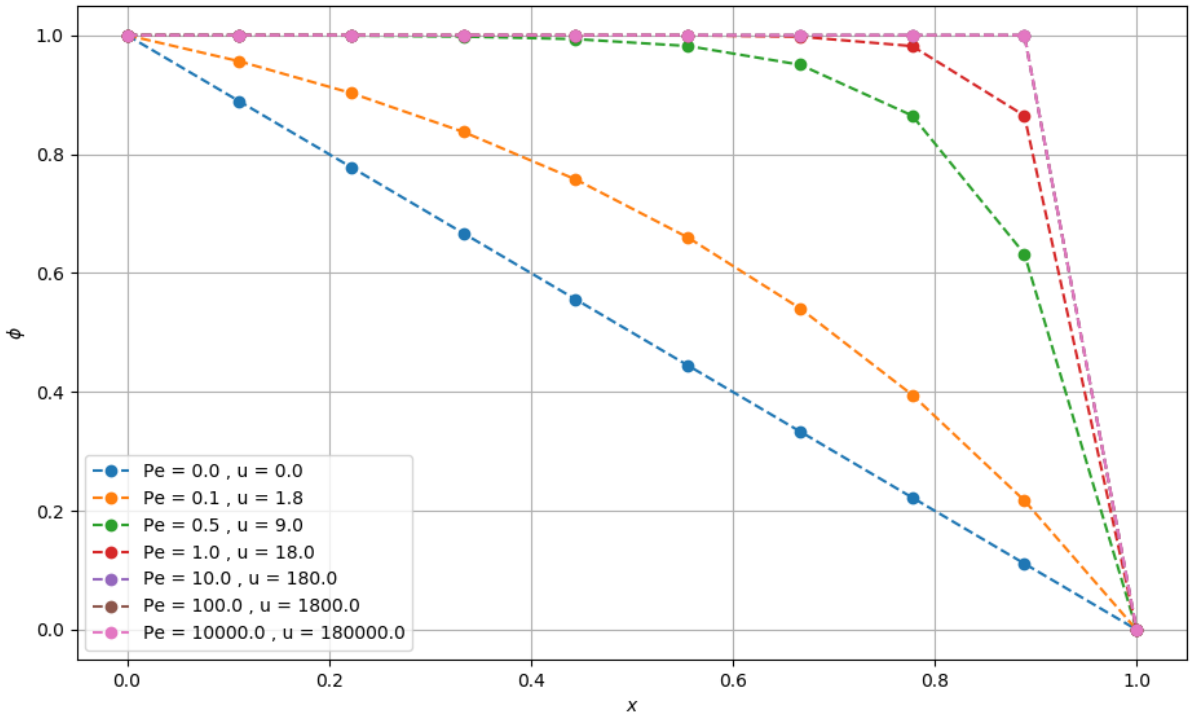

Figure 6: Plot displaying the Convection-Diffusion Solution using the Balancing-Diffusion Approach over a range of Peclet numbers.

Figure 6 displays the results obtained for the FEM Balancing-Diffusion code. The plot displays the ϕ value along the length of the bar. For this example, the same input values used for the Standard-Galerkin Approach:

- $k = 1$
- $L = 1$
- Nodes $= 10$
- $\phi(0) = 1$ *(B.C)*
- $\phi(L) = 0$ *(B.C)*

The code method for the Balancing-Diffusion is like Standard-Galerkin method as u velocity value is calculated from the Pe value. Unlike figure 4, which displays the solution using the standard Galerkin method, there are no oscillations that occur when $Pe > 1$. From the ranges of $0 \leq Pe \leq 1$, the value of ϕ is the same with corresponding Pe values in figure. But as there no oscillations that occur, the values of ϕ are stable unlike figure 1. Furthermore, when the Pe is greater than 1, the solution starts to converge. For the ϕ solution calculated from $Pe > 1$, the nodal value is equal to 1 ($\phi = 1$), until $x = L$, where the solution is 0 ($\phi = 0$). The value is 0, as it is enforced by the boundary conditions.

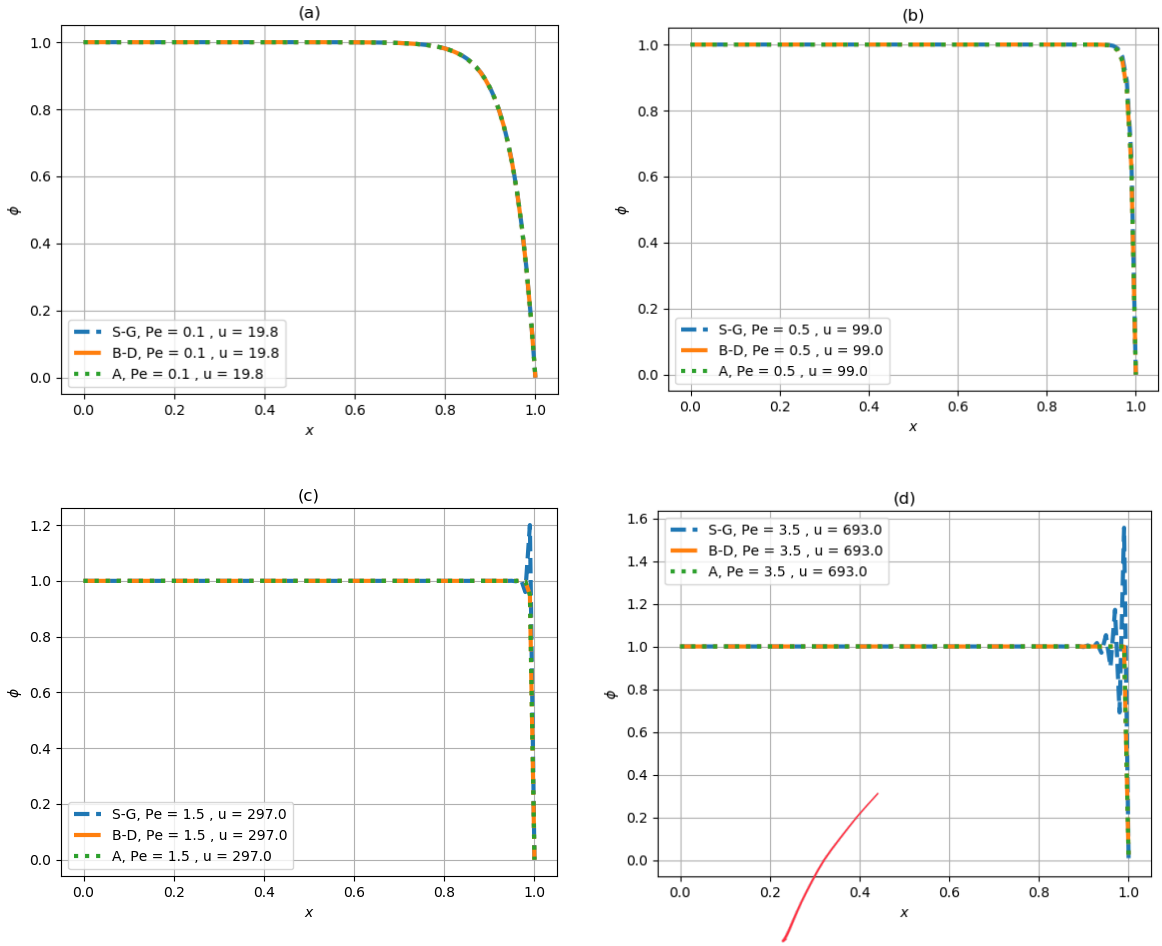

Figure 7 (a-d): Plot Comparison between the Standard-Galerkin (S-G), Balancing-Diffusion (B-D) and Analytical (A) Solutions

Figure 7 displays the solutions to the three different Convection-Diffusion approaches. For the plots, 50 nodes were used. As displayed in figure 7a and 7b, the solutions are three solutions are approximately identical with each other. However, as the Pe value increases past 1, oscillations can be seen in both figure 7c and 7d. The oscillations are greater as the Pe value increases. The Balancing-Diffusion method stays converged with the analytical solution throughout the four different Pe value solutions. The u scaler value, which is determined from Pe also influences the solution. As this value increases, ϕ is equal to 1 for longer, until the boundary conditions enforce $\phi(L) = 0$.

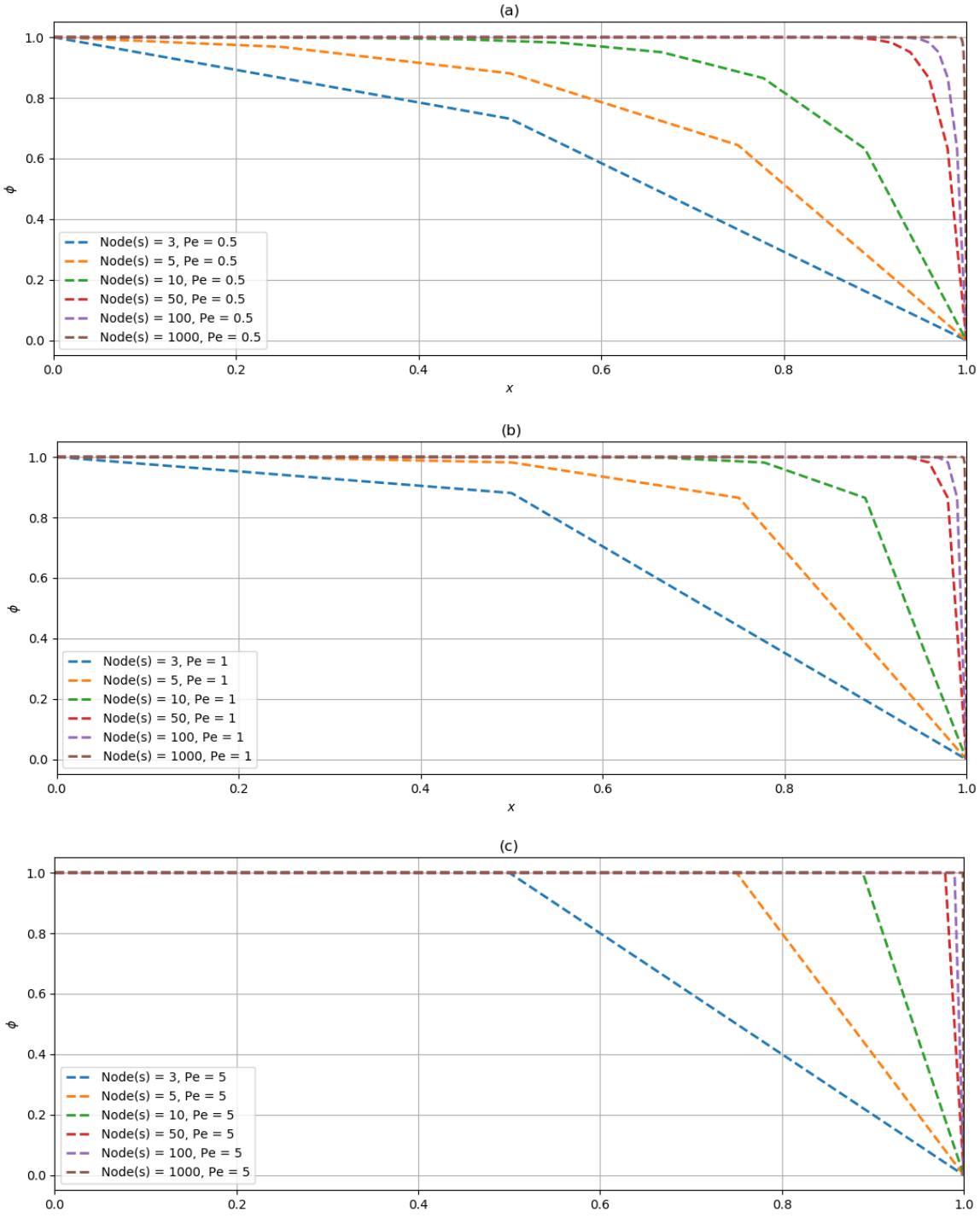

Figure 8 (a-c): Plot displaying the mesh refinement for the Balanced-Diffusion solution at different Peclet values.

Figure 8 displays the influence on the number of nodes and therefore elements used. The solution from figure 8 was computed using the Balancing-Diffusion approach. The input values used for this plot were:

- $k = 1$
- $L = 1$
- $\phi(0) = 1$ *(B.C)*

- $\phi(L) = 0$ *(B.C)*

As the quantity of nodes increase, the ϕ value decay to 0 is delayed further along the length of the bar. For the solution with the $\mathrm{Node(s)} = 1000$, the solution only decays just before $x = L$. Whereas the solution with $\mathrm{Node(s)} = 3$, decays earlier on the length of the bar. Furthermore, the value of ϕ is approximately equal to 1 longer with higher Node quantities. The Pe value influences the plots by changing the value the ϕ, just before it reaches the last Node. For figures 8a and 8b, the value of ϕ is less than 1. However, for plot 8c, the value of ϕ is still approximately one for al nodal amounts, before the final node.

5 Conclusions

Using the Finite Element Method to solve the 1D Convection-Diffusion equation can only produce theoretical similar result when Balancing-Diffusion method is applied for Peclet numbers greater than 1. The Standard-Galerkin Method can only produce solutions that are stable between the range of $0 \leq Pe \leq 1$. As shown in figure 4 and 6, oscillations occur in the Standard Galerkin approach as the solution is not stable. For Peclet numbers greater than 1, the Balancing-Diffusion method is only suitable. Also, the analytical solution produces similar results compared with the Finite Element Method approaches, therefore this proves that the Finite Element Method is a valid numerical approach to solving the Convection-Diffusion equation. Using the plots in figure 8, increasing the quantity of nodes, delays when the ϕ value decreases to 0. Further increasing the Peclet number also increases the ϕ value, ϕ equals 0 which occurs at the last node. Furthermore, using a Peclet number of 0, produces a linear ϕ vector, where the boundary conditions dictate the output.

References

(One reference is used for this project, therefore, no reference note was inserted in the report)

[1] Zienkiewicz, O., Taylor, R., & Nithiarasu, P. (2014). *The finite element method for fluid dynamics / O.C. Zienkiewicz, R.L. Taylor, P. Nithiarasu.* (7th ed.).

Appendix

Code: *(Coded in Python 3.6)*

(Please note: If there are any problems with copying and posting this code into Python, I'd be happy to send the code)

```
#------------------------- Deepak Singh (958840) ------------------------#
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
############################## Minor Functions ##############################


# Local to Global stiffness matrx
def Gather(IN1,OUT1,IND,LOCAL,type_array):
    if type_array == 1: #if == to matrix
        for i in range(0,len(IN1)):
            for j in range(0,len(IN1)):
                LOCAL[OUT1[i],OUT1[j]]  += IND[IN1[i],IN1[j]]
    else: #if not matrix, then vector
```

```
        for i in range(0,len(IN1)):
            LOCAL[OUT1[i]] += IND[IN1[i]]
    return LOCAL


# Python Matrix function (Minus 1 in dictionary)
def Matrix(DICT):
    for i in range(0,len(DICT)):
        DICT[i] = DICT[i]-1
    return DICT


# Calculates Cothx (coshx/sinhx)
def Coth(x):
    if x >= 100: #coshx and sinhx go to infinity
        return float(1)
    else:
        return np.cosh(x)/np.sinh(x)


# Calculates Analytical Solution
def Analytical(Pe,k,L,Nodes):
    x = np.linspace(0,L,Nodes)
    h = L/(Nodes-1)
    u = (Pe*2*k)/h
    return (np.exp((u*x)/k)-np.exp((u*L)/k))/(1-np.exp((u*L)/k))


# Calculates u depending on Pe number
def uscaler(Pe,k,L,Nodes):
    h = L/(Nodes-1)
    return (Pe*2*k)/h


################################## Main Function ##########################
def Main(PeV,Type,k,L,Nodes,Q):


# ------------------------------- Pre-Processing ----------------------#
    h = L/(Nodes-1)
    ndof = 1


    # Boundary Conditions
    phi = []
    for i in range(0,Nodes):
        phi.append("Free")

    phi[0] = 1 #phi @ 0
```

```
    phi[Nodes-1] = 0 #phi @ L

#Type == 1 (Balancing Diffusion) Type >1 && 1<(Standard Galerkin)
    u = uscaler(PeV,k,L,Nodes) #Calculates u scaler depending on Pe number

    # Balancing Diffusion k values computaion
    if Type == 1:

        if PeV == 0:
            PeV = 1E-8
            alpha = Coth(abs(PeV))-(1/abs(PeV)) #Divide by zero warning

        else:
            alpha = Coth(abs(PeV))-(1/abs(PeV))

        kb = 0.5*alpha*u*h
        k+= kb

    # Stiffness and loadvectors
    K = np.zeros((Nodes,Nodes))
    F = np.zeros((Nodes))

#---------------------------- Main Processing-----------------------------#


    for i in range(0,Nodes-1):
        #Generate local stifness matrices
        ke = (u/2)*np.array(([-1,1],[-1,1]))+(k/h)*np.array(([1,-1],[-
1,1]))
        fe = (Q/2)*np.array(([1],[1]))

        #Local Matix ==> Global Matrix
        IN = Matrix({0:1,1:2})
        OUT = {0:1,1:2}

        for j in range(0,len(OUT)):
            OUT[j] = ndof*i+j

        K = Gather(IN,OUT,ke,K,1)
        F = Gather(IN,OUT,fe,F,2)

    A = K
    G = F
    c = 0
    phi_loc = []

    # Condense matrix by enforcing Boundary Condtions and using Gauss ELim
    for i in range(0,Nodes):
        if phi[i] == "Free":
```

```
            phi_loc.append(i)

        else:
            G += -phi[i]*A[:,i-c]
            A = np.delete(A,i-c,axis=0)
            A = np.delete(A,i-c,axis=1)
            G = np.delete(G,i-c,axis=0)
            c += 1

    # Solve missing phi values
    phiv = np.dot(np.linalg.inv(A),G)

    # Replaec phi ca
    c = 0
    for i in range(0,Nodes):
        if i == phi_loc[c]:
            phi[i] = phiv[c]
            c += 1

            if c == len(phi_loc):
                c = 0
    else:
        phi[i] = phi[i]

    return phi

#---------------------------- Post-Processing-----------------------------#
# Figure (1)
#Standard Galerkin discretization is oscillatory beyond an element
#Peclet number ofunity. Plot the scalar variable value at different
#Peclet numbers along the domainlength,

Pe1 = np.array((0,0.1,0.5,1,2,10,100,10000))
k = 1
L = 1
Nodes = 10
x = np.linspace(0,L,Nodes)


for i in range(0,len(Pe1)):
    phi = Main(Pe1[i],0,k,L,Nodes,0)
    plt.figure(1)
    u = uscaler(Pe1[i],k,L,Nodes)
    plt.plot(x,phi,'--o',label='Pe = '+str(Pe1[i])+' , u =
'+str(round(u,2)))

plt.grid(True)
plt.ylabel(r'$\phi$')
plt.xlabel(r'$x$')
```

```
plt.legend()
plt.show()


# Figure (2)
#Stabilized finite element scheme reduces or eliminates spatial
oscillations.
#Plot the scalar variable value at different Peclet numbers along the
#domain length,

Pe2 = np.array((0,0.1,0.5,1,10,100,10000))
k = 1
L = 1
Nodes = 10
x = np.linspace(0,L,Nodes)


for i in range(0,len(Pe2)):
    phi = Main(Pe2[i],1,k,L,Nodes,0)
    u = uscaler(Pe2[i],k,L,Nodes)
    plt.figure(2)
    plt.plot(x,phi,'--o',label='Pe = '+str(Pe2[i])+' , u =
'+str(round(u,2)))

plt.grid(True)
plt.ylabel(r'$\phi$')
plt.xlabel(r'$x$')
plt.legend()
plt.show()


# Figure (3)
Pe = 3.5
k = 1
L = 1
Nodes = 100
x = np.linspace(0,L,Nodes)
phiSG = Main(Pe,2,k,L,Nodes,0)
phiPG = Main(Pe,1,k,L,Nodes,0)
phiA = Analytical(Pe,k,L,Nodes)
u = uscaler(Pe,k,L,Nodes)

plt.figure(3)
plt.plot(x,phiSG,'--',label='S-G, Pe = '+str(Pe)+' , u = '+str(round(u,2)))
plt.plot(x,phiPG,'-.',label='B-D, Pe = '+str(Pe)+' , u = '+str(round(u,2)))
plt.plot(x,phiA,':',label='A, Pe = '+str(Pe)+' , u = '+str(round(u,2)))
plt.grid(True)
plt.ylabel(r'$\phi$')
plt.xlabel(r'$x$')
```

```
plt.legend()
plt.title('(d)')
plt.show()

# Figure (4)
#Plot the stabilized solution for different degrees of mesh refinement,

Pe = 1
Nodes = np.array((3,5,10,50,100,1000))
k = 1
L = 1

for i in range(0,len(Nodes)):
    x = np.linspace(0,L,Nodes[i])
    phi = Main(Pe,1,k,L,Nodes[i],0)
    plt.figure(4)
    plt.plot(x,phi,'--',label='Node(s) = '+str(Nodes[i]))

plt.grid(True)
plt.ylabel(r'$\phi$')
plt.xlabel(r'$x$')
plt.legend()
plt.show()
```