

MANE 6760 (FEM for Fluid Dyn.) Fall 2022: Final Project

1. (10 points) Consider the unsteady, 1D, linear, scalar AD equation. Use the SUPG formulation for linear finite elements and backward Euler time-integration scheme. Set problem parameters as: $a_x = 1.0$, $\kappa = 2.5e - 2$ and $s = 0$. Consider a domain length of $L = 1$, i.e., $x \in [0, L = 1]$, and time duration of $T = L^2/\kappa$, i.e., $t \in [0, T = L^2/\kappa = 40.0]$. Set BCs as: $\phi(x = 0, t) = \phi_0(t)$ and $\phi(x = L, t) = \phi_L(t) = g(t)$, where $g(t) = \min(1.25t/T, 1)$, and IC as: $\phi_{IC}(x, t = 0) = 0$. Use a uniform mesh with $N_e = 10$ elements and a uniform time-step size with $N_t = 50$. Provide the following:

- (a) Provide the plot of the FE solution at $n = 0$ (IC), 10, 20, 30, 40 and 50

The solution plots are attached in Figure 1.

- (b) Provide the Python code

The code to generate these results are in Listing 1.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.linalg import solve_banded
4
5 def get_xmin():
6     # return left end of domain
7     xmin = 0.0
8     return xmin
9
10 def get_xmax():
11     # return right end of domain
12     xmax = 1.0
13     return xmax
14
15 def get_L():
16     # return length of domain
17     L = get_xmax() - get_xmin()
18     return L
19
20 def get_tmin():
21     # return min time
22     tmin = 0.0
23     return tmin
24
25 def get_tmax():
26     # return max time
27     L = get_L()
28     kappa = get_kappa()
29     tmax = (L**2)/kappa
30     return tmax
31
32 def get_T():
33     # return the total time duration
34     T = get_tmax() - get_tmin()
35     return T
36
```

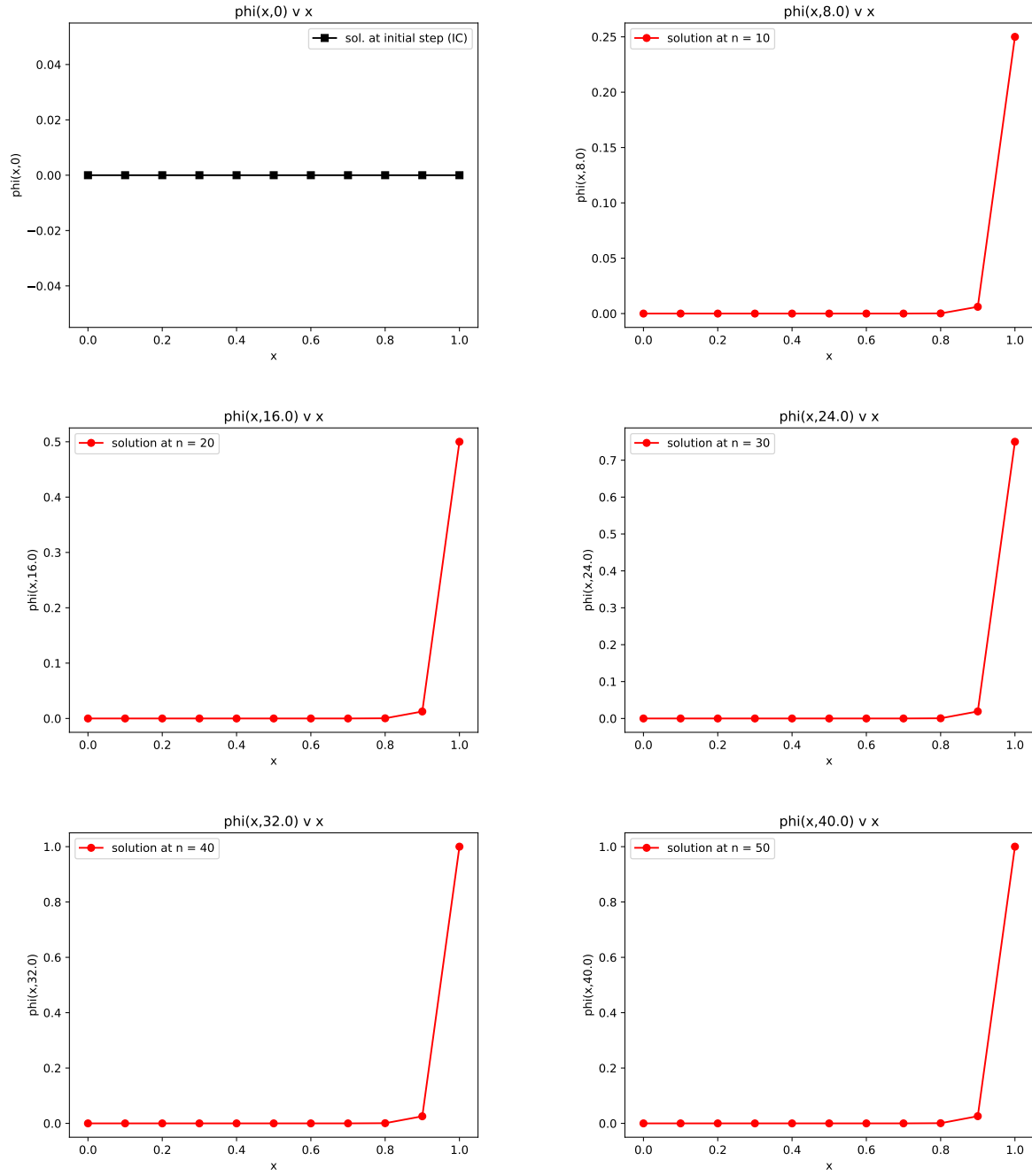


Figure 1: Solution $\phi(x,t)$ at various times

```

37 def get_ax():
38     # return advection velocity value
39     ax = 1.0
40     assert(np.abs(ax)>0)
41     return ax
42
43 def get_kappa():
44     # return kappa value
45     kappa = 2.5e-2
46     assert(kappa>0)
47     return kappa
48
49 def get_Ne():
50     # return number of elements in the mesh
51     Ne = 10
52     assert(Ne>1) # need more than 1 element (otherwise only 2 mesh
53     # vertices for 2 domain end points)
54     return Ne
55
56 def get_Nt():
57     # return number of time intervals
58     Nt = 50 # note number of steps is Nt+1 including t_0 for IC
59     return Nt
60
61 def get_nen():
62     # return number of vertices for an element
63     nen = 2 # 1D
64     return nen
65
66 def get_nes():
67     # return number of shape/basis function for an element
68     nes = 2 # 1D and linear
69     return nes
70
71 def get_neq():
72     # return number of numerical integration/quadrature points for an
73     # element
74     neq = 1 # 1-point rule
75     return neq
76
77 def get_xieq_and_weq():
78     # return location of numerical integration/quadrature points in
79     # parent coordinates of an element
80     neq = get_neq()
81     xieq = np.zeros(neq)
82     xieq[0] = 0.0 # mid-point for 1-point rule in bi-unit 1D element
83     weq = np.zeros(neq)
84     weq[0] = 2.0 # mid-point for 1-point rule in bi-unit 1D element
85     return xieq, weq # mid-point for 1-point rule in bi-unit 1D element
86
87 def get_h():
88     # return mesh size
89     h = get_L()/get_Ne() # uniform mesh
90     return h
91
92 def get_dt():
93     # return time-step size
94     dt = get_T()/get_Nt() # uniform time intervals
95     return dt

```

```

93
94 def get_tau():
95     # return tau value
96     ax = get_ax()
97     kappa = get_kappa()
98     h = get_h()
99     dt = get_dt()
100    tau = 1.0/np.sqrt((2.0/dt)**2 + (2.0*ax/h)**2 + 9.0*(4.0*kappa/(h*h))
101    **2)
102    return tau
103
104 def get_ienarray():
105     # return element-node connectivity
106     Ne = get_Ne()
107     nen = get_nen()
108     ien = np.zeros([Ne,nen])
109     # loop over mesh cells
110     for e in range(Ne): # loop index in [0,Ne-1]
111         ien[e,0] = e
112         ien[e,1] = e+1
113     return ien.astype(int)
114
115 def get_IC():
116     # return IC - 0 at all nodes at t=0
117     Ne = get_Ne()
118     Nn = Ne+1
119     phi_sfem = np.zeros(Nn)
120     return phi_sfem
121
122 def get_left_bdry_value(n):
123     # return left bdry. value (Dirichlet BC)
124     return 0.0
125
126 def get_right_bdry_value(n):
127     tmin = get_tmin()
128     dt = get_dt()
129     T = get_T()
130     t = tmin + n*dt
131     val = (1.25*t)/T
132     if (val < 1.0):
133         return val
134     else:
135         return 1.0
136
137 def get_source(x,n):
138     return 0.0
139
140 def get_shp_and_shpdlcl():
141     # return shape functions and derivatives evaluated at numerical
142     # integration/quadrature points
143     nes = get_nes()
144     neq = get_neq()
145     xieq, weq = get_xieq_and_weq()
146     assert(nes==2) # 1D and linear
147     shp = np.zeros([nes,neq])
148     shpdlcl = np.zeros([nes,neq]) # 1D
149     for q in range(neq): # loop index in [0,neq-1]
150         shp[0,q] = 0.5*(1-xieq[q])
151         shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element

```

```

150         shp[1,q] = 0.5*(1+ xieq[q])
151         shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
152     return shp, shpdlcl
153
154 def apply_num_scheme():
155     # apply numerical scheme
156
157     xmin = get_xmin()
158     xmax = get_xmax()
159
160     tmin = get_tmin()
161     tmax = get_tmax()
162
163     ax = get_ax()
164     kappa = get_kappa()
165
166     Ne = get_Ne()
167     Nn = Ne+1
168     h = get_h()
169
170     Nt = get_Nt()
171     dt = get_dt()
172
173     nen = get_nen()
174     nes = get_nes()
175     neq = get_neq()
176
177     ien = get_ienarray()
178
179     display_phi_plot = True
180
181     tau = get_tau() # constant over mesh when ax, kappa and h are
182                     # constants
183
184     print('Pee:', 0.5*np.abs(ax)*h/kappa) # debug
185     print('CFL:', np.abs(ax)*dt/h)
186     print('tau:', tau) # debug
187
188     kappa_num = tau*ax*ax # constant over mesh when tau and ax are
189                         # constants
190
191     xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh
192                     # vertices
193     tpoints = np.linspace(tmin,tmax,Nt,endpoint=True) # location of time
194                     # points
195
196     # get IC (which satisfies BCs)
197     phi_sfem = get_IC()
198     if (display_phi_plot):
199         plt.plot(xpoints,phi_sfem,'ks-',label='sol. at initial step (IC)'
200         )
201         plt.xlabel('x')
202         plt.ylabel('phi(x,0)')
203         plt.legend(loc='upper right')
204         plt.title('phi(x,0) v x')
205         plt.savefig('Q1_IC.pdf')
206         plt.show()
207     # print(phi_sfem) # debug

```

```

204 xieq, weq = get_xieq_and_weq()
205 shp, shpdlc1 = get_shp_and_shpdlc1() # same type of elements in the
entire mesh
206
207 # loop over time intervals
208 for n in range(1,Nt+1): # loop index [1,Nt]
209     # note 1D and linear elements, and ordered numbering leads to a
tridiagonal banded matrix
210     Kbanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix
including all mesh vertices
211     d = np.zeros(Nn) # right-hand-side vector including all mesh
vertices
212
213     # loop over mesh cells
214     for e in range(Ne): # loop index in [0,Ne-1]
215         # local/element-level data (matrix and vector)
216         assert(nes==nen) # linear elements
217         Ke = np.zeros([nen,nen])
218         Me = np.zeros([nen,nen])
219         Ae = np.zeros([nen,nen])
220         be = np.zeros(nen)
221         de = np.zeros(nen)
222
223         jac = h/2.0 # 1D and linear elements with uniform spacing
224         jacin = 1/jac # 1D and linear elements
225         detj = jac # 1D
226
227         shpdgbl = jacin*shpdlc1
228
229         for q in range(neq): # loop index in [0,neq-1]
230             wdetj = weq[q]*detj
231             phiq = 0.0
232             phidgblq = 0.0
233             xq = 0.0
234
235             for idx_a in range(nes): # loop index in [0,n-1]
236                 phiq = phiq + shp[idx_a,q]*phi_sfem[ien[e,idx_a]]
237                 phidgblq = phidgblq + (shpdgbl[idx_a,q])*phi_sfem[ien
[e,idx_a]]
238                 xq = xq + shp[idx_a,q]*xpoints[ien[e,idx_a]]
239
240             s = get_source(xq,n)
241             for idx_a in range(nes): # loop index in [0,n-1]
242                 be[idx_a] = be[idx_a] + shp[idx_a,q]*s + shpdgbl[
idx_a,q]*tau*ax*s
243
244                 for idx_b in range(nes): # loop index in [0,n-1]
245                     Me[idx_a,idx_b] = Me[idx_a,idx_b] + shp[idx_a,q]*
shp[idx_b,q]*wdetj \
246                                     + shpdgbl[idx_a,q]*(ax*tau)*shp
[idx_b,q]*wdetj
247                     Ae[idx_a,idx_b] = Ae[idx_a,idx_b] - shpdgbl[idx_a
,q]*ax*shp[idx_b,q]*wdetj \
248                                     + shpdgbl[idx_a,q]*(kappa+
kappa_num)*shpdgbl[idx_b,q]*wdetj # ... to be implemented ...
249                     Ke[idx_a,idx_b] = Me[idx_a,idx_b] + dt*Ae[idx_a,
idx_b]
250                     de[idx_a] = de[idx_a] + Me[idx_a,idx_b]*phi_sfem[
ien[e,idx_b]]

```

```

251
252
253     # assembly: recall 1D and linear elements, and ordered
numbering for a tridiagonal matrix
254     for idx_a in range(nes): # loop index in [0,n-1]
255         d[ien[e,idx_a]] = d[ien[e,idx_a]] + de[idx_a]
256         Kbanded[1,ien[e,idx_a]] = Kbanded[1,ien[e,idx_a]] + Ke[
idx_a,idx_a]
257         Kbanded[0,ien[e,1]] = Kbanded[0,ien[e,1]] + Ke[0,1] # upper
side of diagonal
258         Kbanded[2,ien[e,0]] = Kbanded[2,ien[e,0]] + Ke[1,0] # lower
side of diagonal
259
260     # apply BCs
261     phi_sfem[0] = get_left_bdry_value(n) # left BC
262     phi_sfem[Nn-1] = get_right_bdry_value(n) # right BC
263
264     # account for BCs in d
265     d[0] = phi_sfem[0]
266     d[1] = d[1] - Kbanded[2,0]*d[0]
267     d[Nn-1] = phi_sfem[Nn-1]
268     d[Nn-2] = d[Nn-2] - Kbanded[0,Nn-1]*d[Nn-1]
269     Kbanded[1,0] = 1.0
270     Kbanded[0,1] = 0.0 # upper side of diagonal
271     Kbanded[2,0] = 0.0 # lower side of diagonal
272     Kbanded[0,Nn-1] = 0.0 # upper side of diagonal
273     Kbanded[2,Nn-2] = 0.0 # lower side of diagonal
274     Kbanded[1,Nn-1] = 1.0
275
276     phi_sfem = solve_banded((1,1),Kbanded,d)
277
278     if (n%10==0):
279         s = 'solution at n = ' + str(n)
280         s2 = 'Q1_n_'+str(n)+'.pdf'
281         t_npo = get_tmin() + n*get_dt()
282         s3 = 'phi(x,'+str(t_npo)+')'
283         s4 = s3+' v x'
284         if (display_phi_plot):
285             plt.plot(xpoints,phi_sfem,'ro-',label=s)
286             plt.legend(loc='upper left')
287             plt.xlabel('x')
288             plt.ylabel(s3)
289             plt.title(s4)
290             plt.savefig(s2)
291             plt.show()
292
293
294     apply_num_scheme()

```

Listing 1: Transient Advection Diffusion - SUPG stabilized code

2. (40 points) Consider the steady, 1D, non-linear, scalar, ADR equation:

$$\mathcal{L}(\phi) = a_x \phi_x - \kappa \phi_{xx} + c(\phi) \phi = s$$

Use the VMS formulation for linear finite elements such that:

$$\hat{\mathcal{L}}(\cdot) = \mathcal{L}^*(\cdot) = -a_x(\cdot)_{,x} + c(\phi)(\cdot)$$

Set problem parameters as: $a_x = 1, \kappa = 1.0e - 1, c = c_0(1.0 + 0.01\phi), c_0 = 1.0e + 2$ and $s = 10.0$. Consider a domain length of $L = 1$, i.e., $x \in [0, L = 1]$. Set BCs as: $\phi(x = 0) = \phi_0 = 0$ and $\phi(x = 1) = \phi_L = 1.0$. Use a uniform mesh with $N_e = 8$ elements. Set non-linear tolerances to be $1.0e - 6$ and max non-linear iterations to be 100, and take initial guess for the FE solution to be one at *all interior mesh nodes* (and make sure to satisfy BCs). Make use of an appropriate quadrature rule. Derive and provide the following

- (a) **Derive the expression for G_a^e**

Finite Element weak form of the residual can be written as:

$$a(\bar{w}, \bar{\phi}) + a_{stab}(\bar{w}, \bar{\phi}) = 0$$

where,

$$\begin{aligned} a(\bar{w}, \bar{\phi}) &= \int_0^L (-\bar{w}_{,x} a_x \bar{\phi}) + (\bar{w}_{,x} \kappa \bar{\phi}_{,x}) + (\bar{w} c(\bar{\phi}) \bar{\phi}) + (-\bar{w} s) dx \\ a_{stab}(\bar{w}, \bar{\phi}) &= \int_{\hat{\Omega}} (\bar{w}_{,x} a_x \tau a_x \bar{\phi}) + (\bar{w}_{,x} \tau a_x c(\bar{\phi}) \bar{\phi}) + (-\bar{w}_{,x} \tau a_x s) + (-\bar{w} \tau a_x c(\bar{\phi}) \bar{\phi}_{,x}) \\ &\quad + (-\bar{w} \tau c^2(\bar{\phi}) \bar{\phi}) + (\bar{w} \tau c(\bar{\phi}) s) d\hat{\Omega} \end{aligned}$$

This can be split up into terms that have linear terms and terms that do not have linear terms. So, the non-linear residual can now be written as:

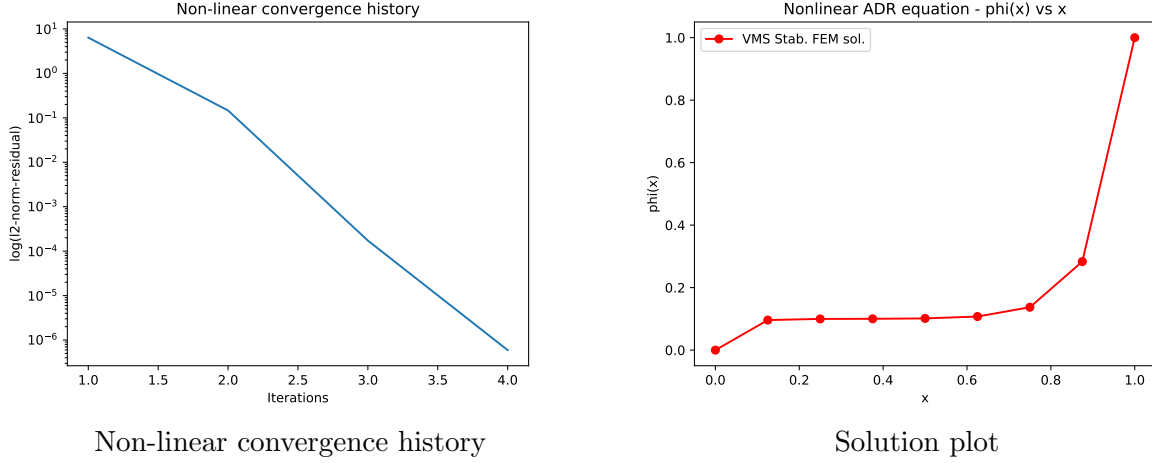
$$\begin{aligned} G_a^e &= G_{a_l}^e + G_{a_{nl}}^e \\ G_{a_l}^e &= \left[\int_{x_l}^{x_r} (-N_{a,x} a_x \bar{\phi}) + (N_{a,x} \kappa \bar{\phi}_{,x}) + (-N_a s) + (N_{a,x} \tau a_x^2 \bar{\phi}_{,x}) + (-N_{a,x} \tau a_x s) dx \right] \\ G_{a_{nl}}^e &= \left[\int_{x_l}^{x_r} \underbrace{N_a c(\bar{\phi}) \bar{\phi}}_I + \underbrace{N_{a,x} \tau a_x c(\bar{\phi}) \bar{\phi}}_{II} + \underbrace{-N_a \tau a_x c(\bar{\phi}) \bar{\phi}_{,x}}_{III} + \underbrace{-N_a \tau c^2(\bar{\phi}) \bar{\phi}}_{IV} + \underbrace{N_a \tau c(\bar{\phi}) s}_{V} dx \right] \end{aligned}$$

- (b) **Derive the expression for $\frac{\partial G_a^e}{\partial \hat{\phi}_b^e}$**

Expression for linearization is:

$$\frac{\partial G_a^e}{\partial \hat{\phi}_b^e} = \frac{\partial}{\partial \hat{\phi}_b^e} (G_{a_l}^e + G_{a_{nl}}^e)$$

$$\begin{aligned} \frac{\partial G_{a_l}^e}{\partial \hat{\phi}_b^e} &= \int_{x_l}^{x_r} (-N_{a,x} a_x N_b) + (N_{a,x} \kappa N_{b,x}) - \mathbf{0} + (N_{a,x} \tau a_x^2 N_{b,x}) - \mathbf{0} dx \\ \frac{\partial G_{a_{nl}}^e}{\partial \hat{\phi}_b^e} &= \frac{\partial}{\partial \hat{\phi}_b^e} (I + II + III + IV + V) \end{aligned}$$



```

cii-wl-40:Final Project vignesh$ python3 final_prob2.py
NL iter (starting at 0): 0
l2 norm of non-linear weak residual: 6.377911401172288
max. nodal value of update (abs. value): 0.9231276899949415
NL iter (starting at 0): 1
l2 norm of non-linear weak residual: 0.14629173712445132
max. nodal value of update (abs. value): 0.02257952862055508
NL iter (starting at 0): 2
l2 norm of non-linear weak residual: 0.0001729513071683056
max. nodal value of update (abs. value): 4.4093477026751724e-05
NL iter (starting at 0): 3
l2 norm of non-linear weak residual: 5.936284757502138e-07
Converged (for NL weak residual)

```

Figure 2: Q2 solutions

$$\begin{aligned}
\frac{\partial I}{\partial \hat{\phi}_b^e} &= \int_{x_l}^{x_r} N_a \bar{\phi} \left(\frac{\partial c(\bar{\phi})}{\partial \bar{\phi}} \right) N_b + N_a c(\bar{\phi}) N_b \, dx \\
\frac{\partial II}{\partial \hat{\phi}_b^e} &= \int_{x_l}^{x_r} N_{a,x} \tau a_x \bar{\phi} \left(\frac{\partial c(\bar{\phi})}{\partial \bar{\phi}} \right) N_b + N_{a,x} \tau a_x c(\bar{\phi}) N_b \, dx \\
\frac{\partial III}{\partial \hat{\phi}_b^e} &= - \left[\int_{x_l}^{x_r} N_a \tau a_x \bar{\phi}_{,x} \left(\frac{\partial c(\bar{\phi})}{\partial \bar{\phi}} \right) N_b + N_a \tau a_x c(\bar{\phi}) N_{b,x} \, dx \right] \\
\frac{\partial IV}{\partial \hat{\phi}_b^e} &= - \left[\int_{x_l}^{x_r} N_a \tau \bar{\phi} (2c(\bar{\phi})) \left(\frac{\partial c(\bar{\phi})}{\partial \bar{\phi}} \right) N_b + N_a \tau c^2(\bar{\phi}) N_b \, dx \right] \\
\frac{\partial V}{\partial \hat{\phi}_b^e} &= \int_{x_l}^{x_r} N_a \tau s \left(\frac{\partial c(\bar{\phi})}{\partial \bar{\phi}} \right) N_b \, dx
\end{aligned}$$

- (c) Provide the non-linear convergence history as well as the plot of the FE solution
The solution and the non-linear convergence history is attached in Figure 2.

- (d) Provide the Python code
The code is attached in Listing 2

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.linalg import solve_banded
4
5 def get_xmin():
6     # return left end of domain

```

```

7     xmin = 0.0
8     return xmin
9
10    def get_xmax():
11        # return right end of domain
12        xmax = 1.0
13        return xmax
14
15    def get_L():
16        # return length of domain
17        L = get_xmax()-get_xmin()
18        return L
19
20    def get_ax():
21        # return advection velocity value
22        ax = 1.0e-0
23        assert(np.abs(ax)>0)
24        return ax
25
26    def get_kappa():
27        # return kappa value
28        kappa = 1.0e-1
29        assert(kappa > 0.0)
30        return kappa
31
32    def get_c0():
33        # return c0 value
34        c0 = 1.0e2
35        return c0
36
37    def get_c(phi):
38        # return c value
39        c0 = get_c0()
40        c = c0*(1.0 + 0.01*phi)
41        ax = get_ax()
42        kappa = get_kappa()
43        assert(ax*ax+4*kappa*c>=0)
44        return c
45
46    def get_dc_dphi(phi):
47        # return dc_dphi value:  $\frac{\partial c}{\partial \phi}$ 
48        dcdphi = (get_c0())*0.01
49        return dcdphi
50
51    def get_source():
52        # return s value
53        s = 10.0
54        return s
55
56    def get_Ne():
57        # return number of elements in the mesh
58        Ne = 8
59        assert(Ne>1) # need more than 1 element (otherwise only 2 mesh
60                    # vertices for 2 domain end points)
61        return Ne
62
63    def get_nen():
64        # return number of vertices for an element
65        nen = 2 # 1D

```

```

65     return nen
66
67 def get_nes():
68     # return number of shape/basis function for an element
69     nes = 2 # 1D and linear
70     return nes
71
72 def get_neq():
73     # return number of numerical integration/quadrature points for an
    element
74     # integrates upto 9th order polynomial accurately
75     neq = 5 # 5-point rule
76     return neq
77
78 def get_xieq_and_weq():
79     # return location of numerical integration/quadrature points in
    parent coordinates of an element
80     neq = get_neq()
81     assert(neq==5)
82     xieq = np.zeros(neq)
83     xieq[0] = (-1.0/3.0)*np.sqrt( 5.0 + 2.0*np.sqrt( 10.0/7.0 ) )
84     xieq[1] = (-1.0/3.0)*np.sqrt( 5.0 - 2.0*np.sqrt( 10.0/7.0 ) )
85     xieq[2] = 0.0
86     xieq[3] = (1.0/3.0)*np.sqrt( 5.0 - 2.0*np.sqrt( 10.0/7.0 ) )
87     xieq[4] = (1.0/3.0)*np.sqrt( 5.0 + 2.0*np.sqrt( 10.0/7.0 ) )
88     weq = np.zeros(neq)
89     weq[0] = ( 322.0 - 13.0*np.sqrt(70.0) )/900.0
90     weq[1] = ( 322.0 + 13.0*np.sqrt(70.0) )/900.0
91     weq[2] = 128.0/225.0
92     weq[3] = ( 322.0 + 13.0*np.sqrt(70.0) )/900.0
93     weq[4] = ( 322.0 - 13.0*np.sqrt(70.0) )/900.0
94     return xieq, weq
95
96 def get_h():
97     # return mesh size
98     h = get_L()/get_Ne() # uniform mesh
99     return h
100
101 def get_tau(c):
102     # return tau alg1 value
103     ax = get_ax()
104     h = get_h()
105     kappa = get_kappa()
106     tau = 1.0/np.sqrt((2.0*ax/h)**2 + 9.0*(4.0*kappa/(h*h))**2 + c**2)
107
108     return tau
109
110 def get_ienarray():
111     # return element-node connectivity
112     Ne = get_Ne()
113     nen = get_nen()
114     ien = np.zeros([Ne,nen])
115     # loop over mesh cells
116     for e in range(Ne): # loop index in [0,Ne-1]
117         ien[e,0] = e
118         ien[e,1] = e+1
119     return ien.astype(int)
120
121 def get_left_bdry_value():

```

```

122     # return left bdry. value (Dirichlet BC)
123     return 0.0
124
125 def get_right_bdry_value():
126     # return right bdry. value (Dirichlet BC)
127     return 1.0
128
129 def get_shp_and_shpdlcl():
130     # return shape functions and derivatives evaluated at numerical
    integration/quadrature points
131     nes = get_nes()
132     neq = get_neq()
133     xieq, weq = get_xieq_and_weq()
134     assert(nes==2) # 1D and linear
135     shp = np.zeros([nes,neq])
136     shpdlcl = np.zeros([nes,neq]) # 1D
137     for q in range(neq): # loop index in [0,neq-1]
138         shp[0,q] = 0.5*(1-xieq[q])
139         shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
140         shp[1,q] = 0.5*(1+ xieq[q])
141         shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
142     return shp, shpdlcl
143
144 def apply_num_scheme():
145     # apply numerical scheme
146
147     xmin = get_xmin()
148     xmax = get_xmax()
149
150     ax = get_ax()
151     kappa = get_kappa()
152     s = get_source() # constant source term
153
154     Ne = get_Ne()
155     Nn = Ne+1
156     h = get_h()
157
158     nen = get_nen()
159     nes = get_nes()
160     neq = get_neq()
161     assert(nes==nen) # linear elements
162
163     ien = get_ienarray()
164
165     display_phi_plot = True
166
167     xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh
    vertices
168
169     xieq, weq = get_xieq_and_weq()
170     shp, shpdlcl = get_shp_and_shpdlcl() # same type of elements in the
    entire mesh
171
172     phi_sfem = np.ones(Nn) # initial guess
173     # apply Dirichlet/essential BCs to initial guess
174     phi_sfem[0] = get_left_bdry_value() # left BC
175     phi_sfem[Nn-1] = get_right_bdry_value() # right BC
176
177     NLmaxiters = 100 # num. of NL max iterations

```

```

178     NLtol = 1.0e-6 # tol. for NL weak residual
179     max_abs_phi_del_tol = 1.0e-6 # tol. for max of absolute value of phi
del/update
180     norm_val1 = []
181     norm_val2 = []
182     iter = []
183     converged_flag = 0
184     # loop over NL iterations
185     for k in range(NLmaxiters): # loop index in [0,NLmaxiters-1]
186
187         iter.append(k+1)
188         # note 1D and linear elements, and ordered numbering leads to a
tridiagonal banded matrix
189         Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix
including all mesh vertices
190         b = np.zeros(Nn) # right-hand-side vector including all mesh
vertices
191
192         # loop over mesh cells
193         for e in range(Ne): # loop index in [0,Ne-1]
194             # local/element-level data (matrix and vector)
195
196             Ae = np.zeros([nen,nen])
197             be = np.zeros(nen)
198
199             jac = h/2.0 # 1D and linear elements with uniform spacing
200             jacin = 1/jac # 1D and linear elements
201             detj = jac # 1D
202
203             shpdgbl = jacin*shpdlcl
204
205             for q in range(neq): # loop index in [0,neq-1]
206                 wdetj = weq[q]*detj
207
208                 phiq = 0.0
209                 phidgblq = 0.0
210
211                 for idx_a in range(nes): # loop index in [0,n-1]
212                     phiq = phiq + shp[idx_a,q]*phi_sfem[ien[e,idx_a]]
213                     phidgblq = phidgblq + (shpdgbl[idx_a,q])*phi_sfem[ien
[e,idx_a]]
214
215                     cq = get_c(phiq)
216                     dcdphiq = get_dc_dphi(phiq)
217                     tauq = get_tau(cq)
218                     kappa_numq = tauq*ax*ax
219                     for idx_a in range(nes): # loop index in [0,n-1]
220                         be[idx_a] = be[idx_a] \
221                             - (shpdgbl[idx_a,q])*ax*phiq*wdetj \
222                             + (shpdgbl[idx_a,q])*(kappa+kappa_numq)*
phidgblq)*wdetj \
223                             - (shp[idx_a,q])*s*wdetj \
224                             - (shpdgbl[idx_a,q])*tauq*ax*s*wdetj \
225                             + (shp[idx_a,q])*cq*phiq*wdetj \
226                             + (shpdgbl[idx_a,q])*tauq*ax*cq*phiq*
wdetj \
227                             - (shp[idx_a,q])*tauq*ax*cq*phidgblq*
wdetj \
228                             - (shp[idx_a,q])*tauq*cq*cq*phiq*wdetj \

```

```

229         + (shp[idx_a,q])*tauq*cq*s*wdetj
230
231         for idx_b in range(nes): # loop index in [0,n-1]
232             Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
233                 - (shpdgbl[idx_a,q])*ax*(shp[
234                 idx_b,q])*wdetj \
235                 + (shpdgbl[idx_a,q])*(kappa+
236                 kappa_numq)*(shpdgbl[idx_b,q])*wdetj \
237                 + (shp[idx_a,q])*(phiq*dcdphiq
238                 + cq)*(shp[idx_b,q])*wdetj \
239                 + (shpdgbl[idx_a,q])*tauq*ax*(
240                 phiq*dcdphiq + cq)*(shp[idx_b,q])*wdetj \
241                 - ((shp[idx_a,q])*tauq*ax*(
242                 phidgblq*dcdphiq)*(shp[idx_b,q]) + (shp[idx_a,q])*tauq*ax*cq*(shpdgbl[
243                 idx_b,q]))*wdetj \
244                 - (shp[idx_a,q])*tauq*(2.0*phiq
245                 *cq*dcdphiq + cq*cq)*(shp[idx_b,q])*wdetj \
246                 + (shp[idx_a,q])*tauq*s*dcdphiq
247                 *(shp[idx_b,q])*wdetj
248
249         # assembly: recall 1D and linear elements, and ordered
250         numbering for a tridiagonal matrix
251         for idx_a in range(nes): # loop index in [0,n-1]
252             b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
253             Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[
254             idx_a,idx_a]
255             Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper
256             side of diagonal
257             Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower
258             side of diagonal
259
260         # account for BCs in b
261         # for now we assume Dirichlet BCs are zero (on left and right
262         ends of the domain)
263         b[0] = 0.0
264         b[Nn-1] = 0.0
265         Abanded[1,0] = 1.0
266         Abanded[0,1] = 0.0 # upper side of diagonal
267         Abanded[2,0] = 0.0 # lower side of diagonal
268         Abanded[0,Nn-1] = 0.0 # upper side of diagonal
269         Abanded[2,Nn-2] = 0.0 # lower side of diagonal
270         Abanded[1,Nn-1] = 1.0
271
272         print('NL iter (starting at 0):',k)
273         NLweak_res_l2 = np.linalg.norm(b)
274         norm_val1.append(NLweak_res_l2)
275         print('l2 norm of non-linear weak residual:',NLweak_res_l2)
276         if (NLweak_res_l2<=NLtol):
277             converged_flag = 1
278             print('Converged (for NL weak residual)')
279             break;
280
281         phi_sfem_del = solve_banded((1,1),Abanded,-b) # note minus sign
282         with 'b' as in '-b'
283
284         max_abs_phi_sfem_del = np.max(np.abs(phi_sfem_del))
285         norm_val2.append(max_abs_phi_sfem_del)
286         print('max. nodal value of update (abs. value):',

```

```

max_abs_phi_sfem_del) # debug
274     if (max_abs_phi_sfem_del <= max_abs_phi_del_tol):
275         converged_flag = 2
276         print('Converged (for update)')
277         break;
278
279     # apply update
280     phi_sfem = phi_sfem + phi_sfem_del
281
282     if (display_phi_plot):
283         plt.plot(xpoints, phi_sfem, 'ro-', label='VMS Stab. FEM sol.')
284         plt.legend(loc='upper left')
285         plt.xlabel('x')
286         plt.ylabel('phi(x)')
287         plt.title('Nonlinear ADR equation - phi(x) vs x')
288         plt.savefig('Q2.pdf')
289         plt.show()
290
291     plt.semilogy(iter, norm_val1)
292     plt.xlabel('Iterations')
293     plt.ylabel('log(l2-norm-residual)')
294     plt.title('Non-linear convergence history')
295     plt.savefig('Q2-convergence.pdf')
296     plt.show()
297
298 apply_num_scheme()

```

Listing 2: Steady, Non-linear ADR equation