

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

In order to execute this project, we need to choose the appropriate mesh for each part of the question and then execute the python script.

1. (20 points) Consider the Python code provided in the course for the stabilized finite element (FE) method for steady, 1D, linear, scalar AD equation. Use the following stabilization parameter: $\tau_{exact1} = \frac{h}{2|a_x|} \left(\frac{1+e^{-2.0Pe^e}}{1-e^{-2.0Pe^e}} - \frac{1}{Pe} \right)$. Consider the following meshes:

The code written in Python is added 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_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-4
29     assert(kappa>0)
30     return kappa
31
32 def get_Ne():
33     # return number of elements in the mesh
34     Ne = get_length()-1
35     assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices
36     # for 2 domain end points)
37     return Ne
38
39 def get_nen():
40     # return number of vertices for an element
41     nen = 2 # 1D
```

```

41     return nen
42
43 def get_nes():
44     # return number of shape/basis function for an element
45     nes = 2 # 1D and linear
46     return nes
47
48 def get_neq():
49     # return number of numerical integration/quadrature points for an element
50     neq = 2 # 1-point rule
51     return neq
52
53 def get_xieq_and_weq():
54     # return location of numerical integration/quadrature points in parent
    coordinates of an element
55     # neq = get_neq()
56     # xieq = np.zeros(neq)
57     # xieq[0] = 0.0 # mid-point for 1-point rule in bi-unit 1D element
58     # weq = np.zeros(neq)
59     # weq[0] = 2.0 # mid-point for 1-point rule in bi-unit 1D element
60     # return xieq, weq # mid-point for 1-point rule in bi-unit 1D element
61     neq = get_neq()
62     assert(neq==2)
63     xieq = np.zeros(neq)
64     xieq[0] = -1.0/np.sqrt(3.0)
65     xieq[1] = 1.0/np.sqrt(3.0)
66     weq = np.zeros(neq)
67     weq[0] = 1.0
68     weq[1] = 1.0
69     return xieq, weq
70
71 def get_xpoints():
72     # xpoints = np.array([0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0])
73     # xpoints = np.array([0.0,0.5,0.75,0.875,1.0])
74     xpoints = np.array([0.0,0.875,1.0])
75     return xpoints
76
77 def get_node(i):
78     xpoints = get_xpoints()
79     return xpoints[i]
80
81 def get_length():
82     xpoints = get_xpoints()
83     return xpoints.size
84
85 def get_h(e):
86     # return mesh size
87     # h = get_L()/get_Ne() # uniform mesh
88     h = get_node(e+1) - get_node(e)
89     return h
90
91
92 def get_tau(e):
93     # return tau value
94     Pee = 0.5*(np.abs(get_ax())*get_h(e))/get_kappa()
95     em2Pee = np.exp(-2.0*Pee)
96     cothPee = (1+em2Pee)/(1-em2Pee)
97     tau = 0.5*(get_h(e)/np.abs(get_ax()))*(cothPee-1.0/Pee)
98     return tau

```

```

99
100 def get_ienarray():
101     # return element-node connectivity
102     Ne = get_Ne()
103     nen = get_nen()
104     ien = np.zeros([Ne,nen])
105     # loop over mesh cells
106     for e in range(Ne): # loop index in [0,Ne-1]
107         ien[e,0] = e
108         ien[e,1] = e+1
109     return ien.astype(int)
110
111 def get_left_bdry_value():
112     # return left bdry. value (Dirichlet BC)
113     return 0.0
114
115 def get_right_bdry_value():
116     # return right bdry. value (Dirichlet BC)
117     return 1.0
118
119 def get_shp_and_shpdlcl():
120     # return shape functions and derivatives evaluated at numerical
121     # integration/quadrature points
122     nes = get_nes()
123     neq = get_neq()
124     xieq, weq = get_xieq_and_weq()
125     assert(nes==2) # 1D and linear
126     shp = np.zeros([nes,neq])
127     shpdlcl = np.zeros([nes,neq]) # 1D
128     for q in range(neq): # loop index in [0,neq-1]
129         shp[0,q] = 0.5*(1-xieq[q])
130         shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
131         shp[1,q] = 0.5*(1+ xieq[q])
132         shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
133     return shp, shpdlcl
134
135 def apply_num_scheme():
136     # apply numerical scheme
137
138     xmin = get_xmin()
139     xmax = get_xmax()
140
141     ax = get_ax()
142     kappa = get_kappa()
143
144     Ne = get_Ne()
145     Nn = Ne+1
146
147     nen = get_nen()
148     nes = get_nes()
149     neq = get_neq()
150
151     ien = get_ienarray()
152
153     display_phi_plot = True
154
155     # xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh
156     # vertices
157     xpoints = get_xpoints()

```

```

156 phi_sfem = np.zeros(Nn)
157
158
159 # note 1D and linear elements, and ordered numbering leads to a
tridiagonal banded matrix
160 Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including
all mesh vertices
161 b = np.zeros(Nn) # right-hand-side vector including all mesh vertices
162
163 # apply BCs
164 phi_sfem[0] = get_left_bdry_value() # left BC
165 phi_sfem[Nn-1] = get_right_bdry_value() # right BC
166
167 xieq, weq = get_xieq_and_weq()
168 shp, shpdlcl = get_shp_and_shpdlcl() # same type of elements in the entire
mesh
169
170 # loop over mesh cells
171 for e in range(Ne): # loop index in [0,Ne-1]
172     h = get_h(e)
173     tau = get_tau(e) # constant over mesh when ax, kappa and h are
constants
174     kappa_num = tau*ax*ax # constant over mesh when tau and ax are
constants
175     # local/element-level data (matrix and vector)
176     assert(nes==nen) # linear elements
177     Ae = np.zeros([nen,nen])
178     be = np.zeros(nen)
179
180     jac = h/2.0 # 1D and linear elements with uniform spacing
181     jacinv = 1/jac # 1D and linear elements
182     detj = jac # 1D
183
184     shpdgbl = jacinv*shpdlcl
185     s = 0.0;
186     for q in range(neq): # loop index in [0,neq-1]
187         wdetj = weq[q]*detj
188         for idx_a in range(nes): # loop index in [0,neg-1]
189             be[idx_a] = be[idx_a] + s*shp[idx_a,q]*wdetj + s*ax*tau*
shpdgbl[idx_a,q]*wdetj # no source term
190             for idx_b in range(nes): # loop index in [0,neg-1]
191                 Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
192                     - (shpdgbl[idx_a,q])*ax*shp[idx_b,q]*
wdetj \
193                     + (shpdgbl[idx_a,q])*(kappa+kappa_num)*(
shpdgbl[idx_b,q])*wdetj
194
195     # assembly: recall 1D and linear elements, and ordered numbering for a
tridiagonal matrix
196     for idx_a in range(nes): # loop index in [0,neg-1]
197         b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
198         Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[idx_a,idx_a]
199     Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper side of
diagonal
200     Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower side of
diagonal
201
202 # account for BCs in b

```

```

203     # for now we assume Dirichlet BCs are zero (on left and right ends of the
        domain)
204     b[0] = phi_sfem[0]
205     b[1] = b[1] - Abanded[2,0]*b[0]
206     b[Nn-1] = phi_sfem[Nn-1]
207     b[Nn-2] = b[Nn-2] - Abanded[0,Nn-1]*b[Nn-1]
208     Abanded[1,0] = 1.0
209     Abanded[0,1] = 0.0 # upper side of diagonal
210     Abanded[2,0] = 0.0 # lower side of diagonal
211     Abanded[0,Nn-1] = 0.0 # upper side of diagonal
212     Abanded[2,Nn-2] = 0.0 # lower side of diagonal
213     Abanded[1,Nn-1] = 1.0
214     phi_sfem = solve_banded((1,1),Abanded,b)
215
216     if (display_phi_plot):
217         plt.plot(xpoints,phi_sfem,'r*-')
218         plt.xlabel('x')
219         plt.ylabel('phi(x)')
220         plt.title('phi(x) v x - Q1c')
221         plt.savefig('Q1c.pdf')
222         plt.show()
223
224 apply_num_scheme()

```

Listing 1: Python code for Q.1

Settings used for this problem are:

```

1     ax = 1.0
2     kappa = 1e-4
3     phi(x=0) = 0.0
4     phi(x=L) = 1.0
5     s = 0.0
6

```

- (a) $M0$ (uniform): $Ne = 8$ with node locations of: $\{0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0\}$
- (b) $M1$ (non-uniform): $Ne = 4$ with node locations of: $\{0.0,0.5,0.75,0.875,1.0\}$
- (c) $M2$ (non-uniform): $Ne = 2$ with node locations of: $\{0.0,0.875,1.0\}$

The solutions to $M0$, $M1$ and $M2$ mesh resolutions are in Fig 1.

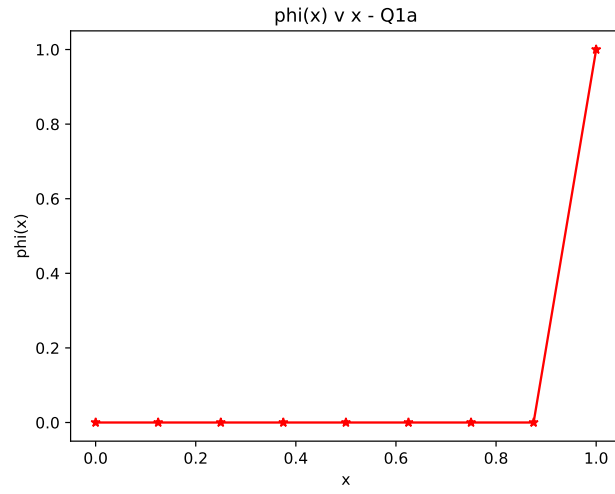
2. (20 points) Consider the Python code provided in the course for the stabilized finite element (FE) method for steady, 1D, linear, scalar AD equation. Use the following stabilization parameter: $\tau_{exact1} = \frac{h}{2|a_x|} \left(\frac{1+e^{-2.0Pe^e}}{1-e^{-2.0Pe^e}} - \frac{1}{Pe^e} \right)$. Set $\phi(x=L) = \phi_L = 0$ and $s = 1.0$. Consider the following meshes:

Listing 2 is the written Python code for this question.

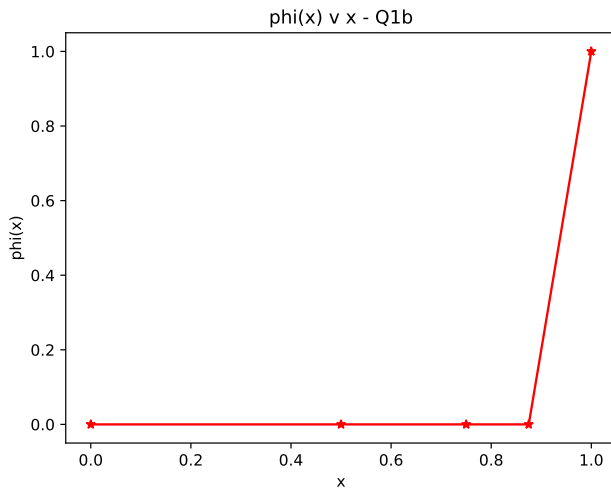
```

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

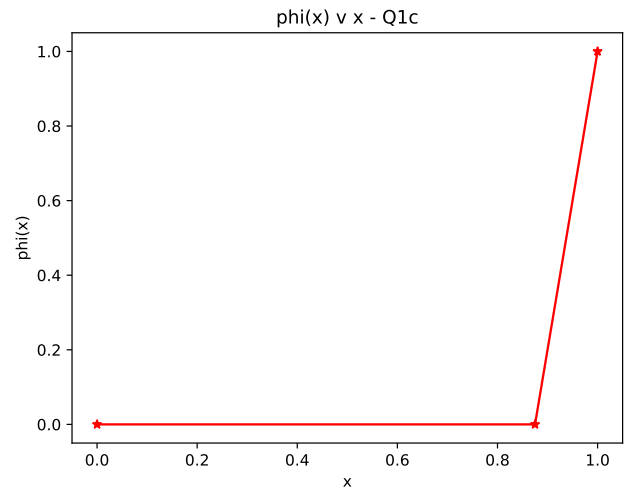
```



(1.a) $M0$



(1.b) $M1$



(1.c) $M2$

Figure 1: Solutions to $M0$ (1.a), $M1$ (1.b) and $M2$ (1.c) meshes

```

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-4
29     assert(kappa>0)
30     return kappa
31
32 def get_Ne():
33     # return number of elements in the mesh
34     #Ne = 100
35     Ne = get_length()-1
36     assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices
37     # for 2 domain end points)
38     return Ne
39
40 def get_nen():
41     # return number of vertices for an element
42     nen = 2 # 1D
43     return nen
44
45 def get_nes():
46     # return number of shape/basis function for an element
47     nes = 2 # 1D and linear
48     return nes
49
50 def get_neq():
51     # return number of numerical integration/quadrature points for an element
52     neq = 2 # 1-point rule
53     return neq
54
55 def get_xieq_and_weq():
56     # return location of numerical integration/quadrature points in parent
57     # coordinates of an element
58     # neq = get_neq()
59     # xieq = np.zeros(neq)
60     # xieq[0] = 0.0 # mid-point for 1-point rule in bi-unit 1D element
61     # weq = np.zeros(neq)
62     # weq[0] = 2.0 # mid-point for 1-point rule in bi-unit 1D element
63     # return xieq, weq # mid-point for 1-point rule in bi-unit 1D element
64     neq = get_neq()
65     assert(neq==2)
66     xieq = np.zeros(neq)
67     xieq[0] = -1.0/np.sqrt(3.0)

```

```

66     xieq[1] = 1.0/np.sqrt(3.0)
67     weq = np.zeros(neq)
68     weq[0] = 1.0
69     weq[1] = 1.0
70     return xieq, weq
71
72 def get_xpoints():
73     xpoints = np.array([0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0])
74     # xpoints = np.array([0.0,0.5,0.75,0.875,1.0])
75     # xpoints = np.array([0.0,0.875,1.0])
76     return xpoints
77
78 def get_node(i):
79     xpoints = get_xpoints()
80     return xpoints[i]
81
82 def get_length():
83     xpoints = get_xpoints()
84     return xpoints.size
85
86 def get_h(e):
87     # return mesh size
88     # h = get_L()/get_Ne() # uniform mesh
89     h = get_node(e+1) - get_node(e)
90     return h
91
92
93 def get_tau(e):
94     # return tau value
95     Pee = 0.5*(np.abs(get_ax())*get_h(e))/get_kappa()
96     em2Pee = np.exp(-2.0*Pee)
97     cothPee = (1+em2Pee)/(1-em2Pee)
98     tau = 0.5*(get_h(e)/np.abs(get_ax()))*(cothPee-1.0/Pee)
99     return tau
100
101 def get_ienarray():
102     # return element-node connectivity
103     Ne = get_Ne()
104     nen = get_nen()
105     ien = np.zeros([Ne,nen])
106     # loop over mesh cells
107     for e in range(Ne): # loop index in [0,Ne-1]
108         ien[e,0] = e
109         ien[e,1] = e+1
110     return ien.astype(int)
111
112 def get_left_bdry_value():
113     # return left bdry. value (Dirichlet BC)
114     return 0.0
115
116 def get_right_bdry_value():
117     # return right bdry. value (Dirichlet BC)
118     return 0.0
119
120 def get_shp_and_shpdlcl():
121     # return shape functions and derivatives evaluated at numerical
    integration/quadrature points
122     nes = get_nes()
123     neq = get_neq()

```



```

124 xieq, weq = get_xieq_and_weq()
125 assert(nes==2) # 1D and linear
126 shp = np.zeros([nes,neq])
127 shpdlcl = np.zeros([nes,neq]) # 1D
128 for q in range(neq): # loop index in [0,neq-1]
129     shp[0,q] = 0.5*(1-xieq[q])
130     shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
131     shp[1,q] = 0.5*(1+ xieq[q])
132     shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
133 return shp, shpdlcl
134
135 def apply_num_scheme():
136     # apply numerical scheme
137
138     xmin = get_xmin()
139     xmax = get_xmax()
140
141     ax = get_ax()
142     kappa = get_kappa()
143
144     Ne = get_Ne()
145     Nn = Ne+1
146
147     nen = get_nen()
148     nes = get_nes()
149     neq = get_neq()
150
151     ien = get_ienarray()
152
153     display_phi_plot = True
154
155     #xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh
vertices
156     xpoints = get_xpoints()
157
158     phi_sfem = np.zeros(Nn)
159
160     # note 1D and linear elements, and ordered numbering leads to a
tridiagonal banded matrix
161     Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including
all mesh vertices
162     b = np.zeros(Nn) # right-hand-side vector including all mesh vertices
163
164     # apply BCs
165     phi_sfem[0] = get_left_bdry_value() # left BC
166     phi_sfem[Nn-1] = get_right_bdry_value() # right BC
167
168     xieq, weq = get_xieq_and_weq()
169     shp, shpdlcl = get_shp_and_shpdlcl() # same type of elements in the entire
mesh
170
171     # loop over mesh cells
172     for e in range(Ne): # loop index in [0,Ne-1]
173         h = get_h(e)
174         tau = get_tau(e) # constant over mesh when ax, kappa and h are
constants
175         kappa_num = tau*ax*ax # constant over mesh when tau and ax are
constants
176

```

```

177     # local/element-level data (matrix and vector)
178     assert(nes==nen) # linear elements
179     Ae = np.zeros([nen,nen])
180     be = np.zeros(nen)
181
182     jac = h/2.0 # 1D and linear elements with uniform spacing
183     jacinv = 1/jac # 1D and linear elements
184     detj = jac # 1D
185     s = 1.0
186     shpdgbl = jacinv*shpdlcl
187
188     for q in range(neq): # loop index in [0,neq-1]
189         wdetj = weq[q]*detj
190         for idx_a in range(nes): # loop index in [0,neg-1]
191             be[idx_a] = be[idx_a] + s*shp[idx_a,q]*wdetj + s*ax*tau*
shpdgbl[idx_a,q]*wdetj # no source term
192             for idx_b in range(nes): # loop index in [0,neg-1]
193                 Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
194                     - (shpdgbl[idx_a,q])*ax*shp[idx_b,q]*
wdetj \
195                     + (shpdgbl[idx_a,q])*(kappa+kappa_num)*(
shpdgbl[idx_b,q])*wdetj
196
197     # assembly: recall 1D and linear elements, and ordered numbering for a
tridiagonal matrix
198     for idx_a in range(nes): # loop index in [0,neg-1]
199         b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
200         Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[idx_a,idx_a]
201
202     Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper side of
diagonal
203     Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower side of
diagonal
204
205     # account for BCs in b
206     # for now we assume Dirichlet BCs are zero (on left and right ends of the
domain)
207     b[0] = phi_sfem[0]
208     b[1] = b[1] - Abanded[2,0]*b[0]
209     b[Nn-1] = phi_sfem[Nn-1]
210     b[Nn-2] = b[Nn-2] - Abanded[0,Nn-1]*b[Nn-1]
211     Abanded[1,0] = 1.0
212     Abanded[0,1] = 0.0 # upper side of diagonal
213     Abanded[2,0] = 0.0 # lower side of diagonal
214     Abanded[0,Nn-1] = 0.0 # upper side of diagonal
215     Abanded[2,Nn-2] = 0.0 # lower side of diagonal
216     Abanded[1,Nn-1] = 1.0
217
218     phi_sfem = solve_banded((1,1),Abanded,b)
219
220     if (display_phi_plot):
221         plt.plot(xpoints,phi_sfem,'r*-')
222         plt.xlabel('x')
223         plt.ylabel('phi(x)')
224         plt.title('phi(x) v x - Q2a')
225         plt.savefig('Q2a.pdf')
226         plt.show()

```

```
227 apply_num_scheme()
```

Listing 2: Python code for Q.2

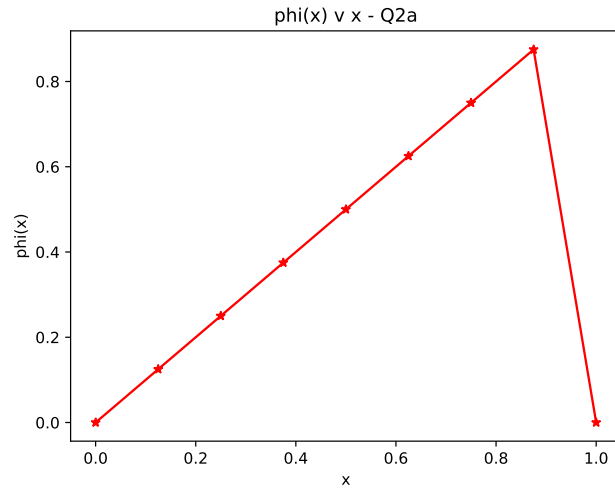
```
1      ax = 1.0
2      kappa = 1e-4
3      phi(x = 0) = 0.0
4      phi(x = L) = 0.0
5      s = 1.0
6
```

- (a) $M0$ (uniform): $N_e = 8$ with node locations of: $\{0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0\}$
- (b) $M1$ (non-uniform): $N_e = 4$ with node locations of: $\{0.0, 0.5, 0.75, 0.875, 1.0\}$
- (c) $M2$ (non-uniform): $N_e = 2$ with node locations of: $\{0.0, 0.875, 1.0\}$

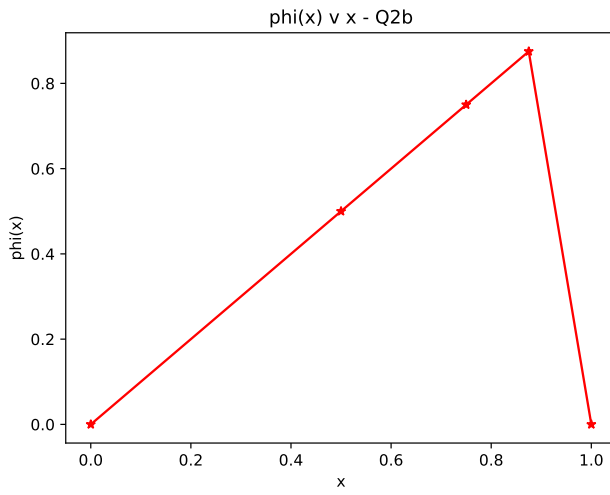
Solutions to $M0$, $M1$ and $M2$ mesh resolutions are in Fig 2.

3. (20 points) Consider the Python code provided in the course for the stabilized finite element (FE) method for steady, 1D, linear, scalar ADR equation. Use the VMS formulation (i.e., $\hat{\mathcal{L}}(\cdot) = \mathcal{L}^*(\cdot)$). Set $\kappa = 1.0e - 1$, $c = 1.0e + 2$ and $s = 10.0$. Consider the following meshes: The Python code for this problem is added in Listing 3.

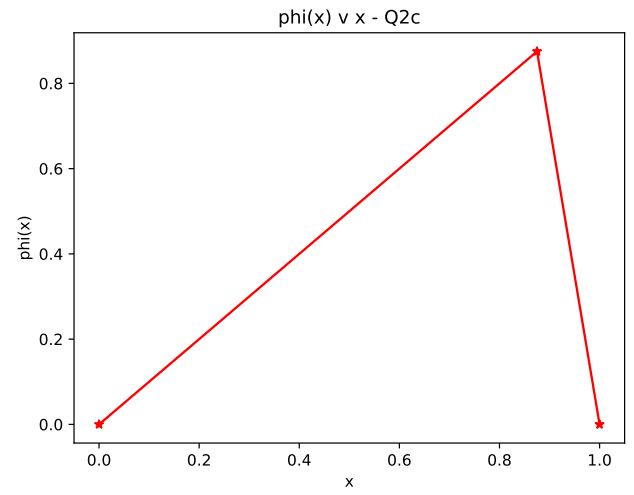
```
1 from webbrowser import get
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.linalg import solve_banded
5
6 def get_xmin():
7     # return left end of domain
8     xmin = 0.0
9     return xmin
10
11 def get_xmax():
12     # return right end of domain
13     xmax = 1.0
14     return xmax
15
16 def get_L():
17     # return length of domain
18     L = get_xmax() - get_xmin()
19     return L
20
21 def get_ax():
22     # return advection velocity value
23     ax = 1.0e-0
24     assert(np.abs(ax) > 0)
25     return ax
26
27 def get_kappa():
28     # return kappa value
29     kappa = 1.0e-1
30     assert(kappa > 0)
31     return kappa
32
33 def get_c():
```



(2.a) $M0$



(2.b) $M1$



(2.c) $M2$

Figure 2: Solutions to $M0$ (2.a), $M1$ (2.b) and $M2$ (2.c) meshes

```

34     # return 'c' or reactivity (c<0 implies production and c>0 implies
    destruction)
35     c = 1.0e+2
36     ax = get_ax()
37     kappa = get_kappa()
38     assert(ax*ax+4*kappa*c>=0)
39     return c
40
41 def get_Ne():
42     # return number of elements in the mesh
43     xp = get_xpoints()
44     Ne = xp.size-1
45     assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices
    for 2 domain end points)
46     return Ne
47
48 def get_nen():
49     # return number of vertices for an element
50     nen = 2 # 1D
51     return nen
52
53 def get_nes():
54     # return number of shape/basis function for an element
55     nes = 2 # 1D and linear
56     return nes
57
58 def get_neq():
59     # return number of numerical integration/quadrature points for an element
60     neq = 2 # 2-point rule
61     return neq
62
63 def get_xieq_and_weq():
64     # return location of numerical integration/quadrature points in parent
    coordinates of an element
65     neq = get_neq()
66     assert(neq==2)
67     xieq = np.zeros(neq)
68     xieq[0] = -1.0/np.sqrt(3.0)
69     xieq[1] = 1.0/np.sqrt(3.0)
70     weq = np.zeros(neq)
71     weq[0] = 1.0
72     weq[1] = 1.0
73     return xieq, weq
74
75 def get_xpoints():
76     xpoints = np.array([0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0])
77     # xpoints = np.array([0.0,0.125,0.25,0.5,0.75,0.875,1.0])
78     return xpoints
79
80 def get_node(i):
81     xpoints = get_xpoints()
82     return xpoints[i]
83
84 def get_h(e):
85     # return mesh size
86     # h = get_L()/get_Ne() # uniform mesh
87     h = get_node(e+1) - get_node(e)
88     return h
89

```

```

90 def get_tau(e):
91     # return tau alg1 value
92     ax = get_ax()
93     kappa = get_kappa()
94     c = get_c()
95     h = get_h(e)
96     tau = 1.0/np.sqrt((2.0*ax/h)**2 + 9.0*(4.0*kappa/(h*h))**2+c*c)
97     return tau
98
99 def get_ienarray():
100     # return element-node connectivity
101     Ne = get_Ne()
102     nen = get_nen()
103     ien = np.zeros([Ne,nen])
104     # loop over mesh cells
105     for e in range(Ne): # loop index in [0,Ne-1]
106         ien[e,0] = e
107         ien[e,1] = e+1
108     return ien.astype(int)
109
110 def get_left_bdry_value():
111     # return left bdry. value (Dirichlet BC)
112     return 0.0
113
114 def get_right_bdry_value():
115     # return right bdry. value (Dirichlet BC)
116     return 1.0
117
118 def get_shp_and_shpdlcl():
119     # return shape functions and derivatives evaluated at numerical
120     # integration/quadrature points
121     nes = get_nes()
122     neq = get_neq()
123     xieq, weq = get_xieq_and_weq()
124     assert(nes==2) # 1D and linear
125     shp = np.zeros([nes,neq])
126     shpdlcl = np.zeros([nes,neq]) # 1D
127     for q in range(neq): # loop index in [0,neq-1]
128         shp[0,q] = 0.5*(1-xieq[q])
129         shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
130         shp[1,q] = 0.5*(1+ xieq[q])
131         shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
132     return shp, shpdlcl
133
134 def apply_num_scheme():
135     # apply numerical scheme
136
137     xmin = get_xmin()
138     xmax = get_xmax()
139
140     ax = get_ax()
141     kappa = get_kappa()
142     c = get_c()
143
144     Ne = get_Ne()
145     Nn = Ne+1
146
147     nen = get_nen()

```

```

148     nes = get_nes()
149     neq = get_neq()
150
151     ien = get_ienarray()
152
153     display_phi_plot = True
154
155     # xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh
vertices
156     xpoints = get_xpoints()
157
158     phi_sfem = np.zeros(Nn)
159
160     # note 1D and linear elements, and ordered numbering leads to a
tridiagonal banded matrix
161     Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including
all mesh vertices
162     b = np.zeros(Nn) # right-hand-side vector including all mesh vertices
163
164     # apply BCs
165     phi_sfem[0] = get_left_bdry_value() # left BC
166     phi_sfem[Nn-1] = get_right_bdry_value() # right BC
167
168     xieq, weq = get_xieq_and_weq()
169     shp, shpdlc1 = get_shp_and_shpdlc1() # same type of elements in the entire
mesh
170
171     # loop over mesh cells
172     for e in range(Ne): # loop index in [0,Ne-1]
173
174         h = get_h(e)
175         tau = get_tau(e) # constant over mesh when ax, kappa, c and h are
constants
176         kappa_num = tau*ax*ax # constant over mesh when tau and ax are
constants
177         ax_num1 = -tau*ax*c # constant over mesh when tau, ax and c are
constants
178         f_stab = 1.0 # f_stab=0.0 for SUPG, f_stab=-1.0 for GLS, or f_stab=1.0
for VMS
179         ax_num2 = -f_stab*tau*c*ax # constant over mesh when tau, ax and c are
constants
180         c_num = -f_stab*tau*c*c # constant over mesh when tau and c are
constants
181
182         # local/element-level data (matrix and vector)
183         assert(nes==nen) # linear elements
184         Ae = np.zeros([nen,nen])
185         be = np.zeros(nen)
186
187         jac = h/2.0 # 1D and linear elements with uniform spacing
188         jacinv = 1/jac # 1D and linear elements
189         detj = jac # 1D
190         s = 10.0
191         shpdgbl = jacinv*shpdlc1
192
193         for q in range(neq): # loop index in [0,neq-1]
194             wdetj = weq[q]*detj
195             for idx_a in range(nes): # loop index in [0,nen-1]
196                 be[idx_a] = be[idx_a] + shp[idx_a,q]*s*wdetj -c*f_stab*tau*s*

```

```

shp[idx_a,q]*wdetj + ax*tau*shpdgbl[idx_a,q]*s*wdetj# source term
197     for idx_b in range(nes): # loop index in [0,n-1]
198         Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
199             - (shpdgbl[idx_a,q])*(ax+ax_num1)*shp[
idx_b,q]*wdetj \
200             + (shpdgbl[idx_a,q])*(kappa+kappa_num)*(
shpdgbl[idx_b,q])*wdetj \
201             + shp[idx_a,q]*ax_num2*(shpdgbl[idx_b,q]
)*wdetj \
202             + shp[idx_a,q]*(c+c_num)*shp[idx_b,q]*
wdetj
203
204     # assembly: recall 1D and linear elements, and ordered numbering for a
tridiagonal matrix
205     for idx_a in range(nes): # loop index in [0,n-1]
206         b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
207         Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[idx_a,idx_a]
]
208     Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper side of
diagonal
209     Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower side of
diagonal
210
211     # account for BCs in b
212     # for now we assume Dirichlet BCs are zero (on left and right ends of the
domain)
213     b[0] = phi_sfem[0]
214     b[1] = b[1] - Abanded[2,0]*b[0]
215     b[Nn-1] = phi_sfem[Nn-1]
216     b[Nn-2] = b[Nn-2] - Abanded[0,Nn-1]*b[Nn-1]
217     Abanded[1,0] = 1.0
218     Abanded[0,1] = 0.0 # upper side of diagonal
219     Abanded[2,0] = 0.0 # lower side of diagonal
220     Abanded[0,Nn-1] = 0.0 # upper side of diagonal
221     Abanded[2,Nn-2] = 0.0 # lower side of diagonal
222     Abanded[1,Nn-1] = 1.0
223
224     phi_sfem = solve_banded((1,1),Abanded,b)
225
226     if (display_phi_plot):
227         plt.plot(xpoints,phi_sfem,'r*-')
228         plt.xlabel('x')
229         plt.ylabel('phi(x)')
230         plt.title('phi(x) v x - Q3a')
231         plt.savefig('Q3a.pdf')
232         plt.show()
233
234 apply_num_scheme()

```

Listing 3: Python code for Q3.

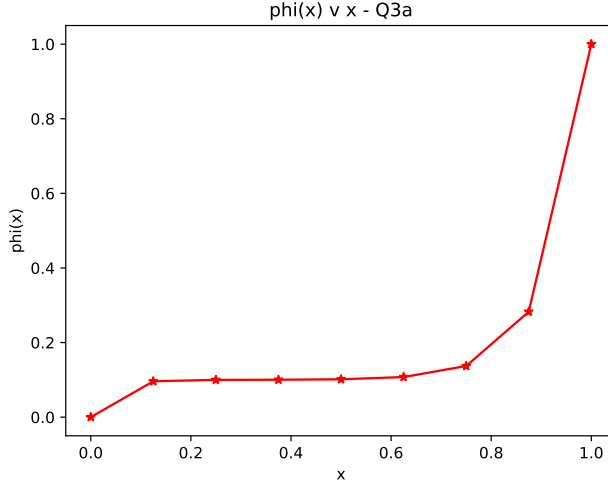
```

1     ax = 1.0
2     kappa = 1.0e-1
3     c = 1.0e+2
4     s = 10.0
5     phi(x = 0) = 0.0
6     phi(x = L) = 1.0
7

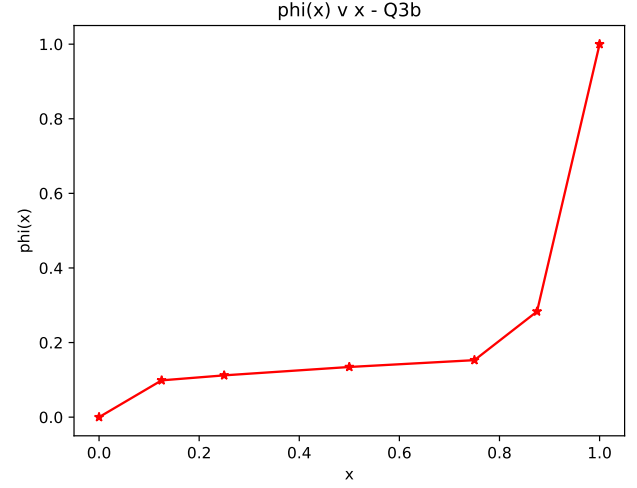
```


- (a) $M0(\text{uniform})$: $N_e = 8$ with node locations of: $\{0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0\}$
- (b) $M1(\text{non-uniform})$: $N_e = 6$ with node locations of: $\{0, 0.125, 0.25, 0.5, 0.75, 0.875, 1.0\}$

Solutions to $M0$ (3.a) and $M1$ (3.b) mesh resolutions are in Fig 3.



(3.a) $M0$



(3.b) $M1$

Figure 3: Solutions to $M0$ (3.a) and $M1$ (3.b)