MANE-6760
Vignesh Ramakrishnan
RIN: 662028006
**Due: 11pm October 27, 2022**

# 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.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve_banded

def get_xmin():
    # return left end of domain
    xmin = 0.0
    return xmin

def get_xmax():
    # return right end of domain
    xmax = 1.0
    return xmax

def get_L():
    # return length of domain
    L = get_xmax()-get_xmin()
    return L

def get_ax():
    # return advection velocity value
    ax = 1.0e-0
    assert(np.abs(ax)>0)
    return ax

def get_kappa():
    # return kappa value
    kappa = 1.0e-4
    assert(kappa>0)
    return kappa

def get_Ne():
    # return number of elements in the mesh
    Ne = get_length()-1
    assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices for 2 domain end points)
    return Ne

def get_nen():
    # return number of vertices for an element
    nen = 2 # 1D
```

```
    return nen

def get_nes():
    # return number of shape/basis function for an element
    nes = 2 # 1D and linear
    return nes

def get_neq():
    # return number of numerical integration/quadrature points for an element
    neq = 2 # 1-point rule
    return neq

def get_xieq_and_weq():
    # return location of numerical integration/quadrature points in parent coordinates of an element
    # neq = get_neq()
    # xieq = np.zeros(neq)
    # xieq[0] = 0.0 # mid-point for 1-point rule in bi-unit 1D element
    # weq = np.zeros(neq)
    # weq[0] = 2.0 # mid-point for 1-point rule in bi-unit 1D element
    # return xieq, weq # mid-point for 1-point rule in bi-unit 1D element
    neq = get_neq()
    assert(neq==2)
    xieq = np.zeros(neq)
    xieq[0] = -1.0/np.sqrt(3.0)
    xieq[1] =  1.0/np.sqrt(3.0)
    weq = np.zeros(neq)
    weq[0] = 1.0
    weq[1] = 1.0
    return xieq, weq

def get_xpoints():
    # xpoints = np.array([0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0])
    # xpoints = np.array([0.0,0.5,0.75,0.875,1.0])
    xpoints = np.array([0.0,0.875,1.0])
    return xpoints

def get_node(i):
    xpoints = get_xpoints()
    return xpoints[i]

def get_length():
    xpoints = get_xpoints()
    return xpoints.size

def get_h(e):
    # return mesh size
    # h = get_L()/get_Ne() # uniform mesh
    h = get_node(e+1) - get_node(e)
    return h


def get_tau(e):
    # return tau value
    Pee = 0.5*(np.abs(get_ax())*get_h(e))/get_kappa()
    em2Pee = np.exp(-2.0*Pee)
    cothPee = (1+em2Pee)/(1-em2Pee)
    tau = 0.5*(get_h(e)/np.abs(get_ax()))*(cothPee-1.0/Pee)
    return tau
```

```python

def get_ienarray():
    # return element-node connectivity
    Ne = get_Ne()
    nen = get_nen()
    ien = np.zeros([Ne,nen])
    # loop over mesh cells
    for e in range(Ne): # loop index in [0,Ne-1]
        ien[e,0] = e
        ien[e,1] = e+1
    return ien.astype(int)

def get_left_bdry_value():
    # return left bdry. value (Dirichlet BC)
    return 0.0

def get_right_bdry_value():
    # return right bdry. value (Dirichlet BC)
    return 1.0

def get_shp_and_shpdlcl():
    # return shape functions and derivatives evaluated at numerical
    integration/quadrature points
    nes = get_nes()
    neq = get_neq()
    xieq, weq = get_xieq_and_weq()
    assert(nes==2) # 1D and linear
    shp = np.zeros([nes,neq])
    shpdlcl = np.zeros([nes,neq]) # 1D
    for q in range(neq): # loop index in [0,neq-1]
        shp[0,q] = 0.5*(1-xieq[q])
        shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
        shp[1,q] = 0.5*(1+ xieq[q])
        shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
    return shp, shpdlcl

def apply_num_scheme():
    # apply numerical scheme

    xmin = get_xmin()
    xmax = get_xmax()

    ax = get_ax()
    kappa = get_kappa()

    Ne = get_Ne()
    Nn = Ne+1

    nen = get_nen()
    nes = get_nes()
    neq = get_neq()

    ien = get_ienarray()

    display_phi_plot = True

    # xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh
    vertices
    xpoints = get_xpoints()
```

```
    phi_sfem = np.zeros(Nn)

    # note 1D and linear elements, and ordered numbering leads to a tridiagonal banded matrix
    Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including all mesh vertices
    b = np.zeros(Nn) # right-hand-side vector including all mesh vertices

    # apply BCs
    phi_sfem[0] = get_left_bdry_value() # left BC
    phi_sfem[Nn-1] = get_right_bdry_value() # right BC

    xieq, weq = get_xieq_and_weq()
    shp, shpdlcl = get_shp_and_shpdlcl() # same type of elements in the entire mesh

    # loop over mesh cells
    for e in range(Ne): # loop index in [0,Ne-1]
        h = get_h(e)
        tau = get_tau(e) # constant over mesh when ax, kappa and h are constants
        kappa_num = tau*ax*ax # constant over mesh when tau and ax are constants
        # local/element-level data (matrix and vector)
        assert(nes==nen) # linear elements
        Ae = np.zeros([nen,nen])
        be = np.zeros(nen)

        jac = h/2.0 # 1D and linear elements with uniform spacing
        jacinv = 1/jac # 1D and linear elements
        detj = jac # 1D

        shpdgbl = jacinv*shpdlcl
        s = 0.0;
        for q in range(neq): # loop index in [0,neq-1]
            wdetj = weq[q]*detj
            for idx_a in range(nes): # loop index in [0,nes-1]
                be[idx_a] = be[idx_a] + s*shp[idx_a,q]*wdetj +  s*ax*tau*shpdgbl[idx_a,q]*wdetj# no source term
                for idx_b in range(nes): # loop index in [0,nes-1]
                    Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
                                      - (shpdgbl[idx_a,q])*ax*shp[idx_b,q]*wdetj \
                                      + (shpdgbl[idx_a,q])*(kappa+kappa_num)*(shpdgbl[idx_b,q])*wdetj

        # assembly: recall 1D and linear elements, and ordered numbering for a tridiagonal matrix
        for idx_a in range(nes): # loop index in [0,nes-1]
            b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
            Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[idx_a,idx_a]
        Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper side of diagonal
        Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower side of diagonal

    # account for BCs in b
```

```python
    # for now we assume Dirichlet BCs are zero (on left and right ends of the domain)
    b[0] = phi_sfem[0]
    b[1] = b[1] - Abanded[2,0]*b[0]
    b[Nn-1] = phi_sfem[Nn-1]
    b[Nn-2] = b[Nn-2] - Abanded[0,Nn-1]*b[Nn-1]
    Abanded[1,0] = 1.0
    Abanded[0,1] = 0.0 # upper side of diagonal
    Abanded[2,0] = 0.0 # lower side of diagonal
    Abanded[0,Nn-1] = 0.0 # upper side of diagonal
    Abanded[2,Nn-2] = 0.0 # lower side of diagonal
    Abanded[1,Nn-1] = 1.0
    phi_sfem = solve_banded((1,1),Abanded,b)

    if (display_phi_plot):
        plt.plot(xpoints,phi_sfem,'r*-')
        plt.xlabel('x')
        plt.ylabel('phi(x)')
        plt.title('phi(x) v x - Q1c')
        plt.savefig('Q1c.pdf')
        plt.show()

apply_num_scheme()
```

Listing 1: Python code for Q.1

Settings used for this problem are:

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

```

(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}

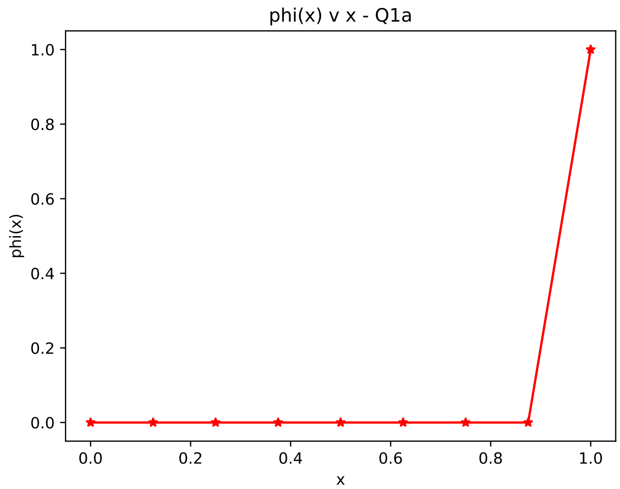
(c) $M2$(non-uniform): $Ne = 2$ with node locations of: {0.0,0.875,1.0}

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

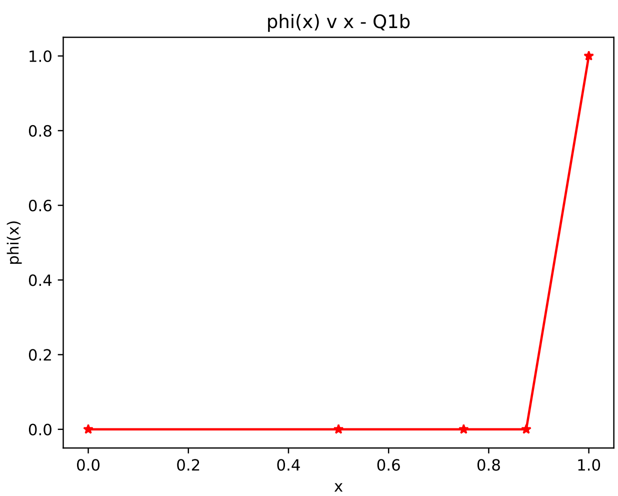
2. (20 points) Consider the Python code provided in the course for the stabilized finite element (FE) method for steady, 1D, linear, scalar AD equation. Use the following stabilization parameter: $\tau_{exact1} = \frac{h}{2|a_x|}\left(\frac{1+e^{-2.0Pe^e}}{1-e^{-2.0Pe^e}} - \frac{1}{Pe}\right)$. Set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

Listing 2 is the written Python code for this question.

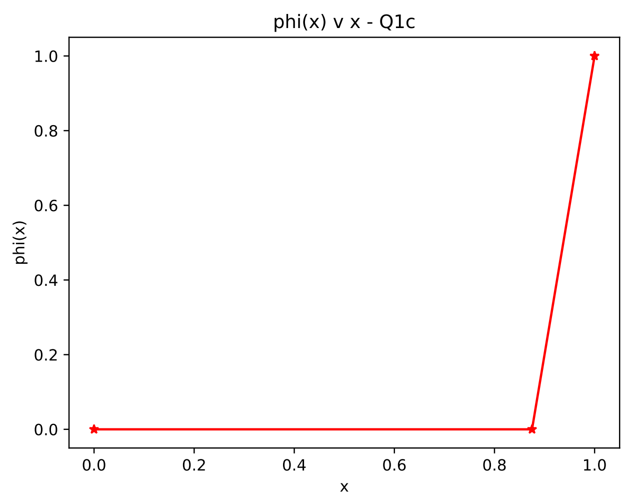
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve_banded

def get_xmin():
    # return left end of domain
    xmin = 0.0
    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

```

def get_xmax():
    # return right end of domain
    xmax = 1.0
    return xmax

def get_L():
    # return length of domain
    L = get_xmax()-get_xmin()
    return L

def get_ax():
    # return advection velocity value
    ax = 1.0e-0
    assert(np.abs(ax)>0)
    return ax

def get_kappa():
    # return kappa value
    kappa = 1.0e-4
    assert(kappa>0)
    return kappa

def get_Ne():
    # return number of elements in the mesh
    #Ne = 100
    Ne = get_length()-1
    assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices
    for 2 domain end points)
    return Ne

def get_nen():
    # return number of vertices for an element
    nen = 2 # 1D
    return nen

def get_nes():
    # return number of shape/basis function for an element
    nes = 2 # 1D and linear
    return nes

def get_neq():
    # return number of numerical integration/quadrature points for an element
    neq = 2 # 1-point rule
    return neq

def get_xieq_and_weq():
    # return location of numerical integration/quadrature points in parent
    coordinates of an element
    # neq = get_neq()
    # xieq = np.zeros(neq)
    # xieq[0] = 0.0 # mid-point for 1-point rule in bi-unit 1D element
    # weq = np.zeros(neq)
    # weq[0] = 2.0 # mid-point for 1-point rule in bi-unit 1D element
    # return xieq, weq # mid-point for 1-point rule in bi-unit 1D element
    neq = get_neq()
    assert(neq==2)
    xieq = np.zeros(neq)
    xieq[0] = -1.0/np.sqrt(3.0)
```

```
    xieq[1] =  1.0/np.sqrt(3.0)
    weq = np.zeros(neq)
    weq[0] = 1.0
    weq[1] = 1.0
    return xieq, weq

def get_xpoints():
    xpoints = np.array([0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0])
    # xpoints = np.array([0.0,0.5,0.75,0.875,1.0])
    # xpoints = np.array([0.0,0.875,1.0])
    return xpoints

def get_node(i):
    xpoints = get_xpoints()
    return xpoints[i]

def get_length():
    xpoints = get_xpoints()
    return xpoints.size

def get_h(e):
    # return mesh size
    # h = get_L()/get_Ne() # uniform mesh
    h = get_node(e+1) - get_node(e)
    return h


def get_tau(e):
    # return tau value
    Pee = 0.5*(np.abs(get_ax())*get_h(e))/get_kappa()
    em2Pee = np.exp(-2.0*Pee)
    cothPee = (1+em2Pee)/(1-em2Pee)
    tau = 0.5*(get_h(e)/np.abs(get_ax()))*(cothPee-1.0/Pee)
    return tau

def get_ienarray():
    # return element-node connectivity
    Ne = get_Ne()
    nen = get_nen()
    ien = np.zeros([Ne,nen])
    # loop over mesh cells
    for e in range(Ne): # loop index in [0,Ne-1]
        ien[e,0] = e
        ien[e,1] = e+1
    return ien.astype(int)

def get_left_bdry_value():
    # return left bdry. value (Dirichlet BC)
    return 0.0

def get_right_bdry_value():
    # return right bdry. value (Dirichlet BC)
    return 0.0

def get_shp_and_shpdlcl():
    # return shape functions and derivatives evaluated at numerical
    integration/quadrature points
    nes = get_nes()
    neq = get_neq()
```

```
    xieq, weq = get_xieq_and_weq()
    assert(nes==2) # 1D and linear
    shp = np.zeros([nes,neq])
    shpdlcl = np.zeros([nes,neq]) # 1D
    for q in range(neq): # loop index in [0,neq-1]
        shp[0,q] = 0.5*(1-xieq[q])
        shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
        shp[1,q] = 0.5*(1+ xieq[q])
        shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
    return shp, shpdlcl

def apply_num_scheme():
    # apply numerical scheme

    xmin = get_xmin()
    xmax = get_xmax()

    ax = get_ax()
    kappa = get_kappa()

    Ne = get_Ne()
    Nn = Ne+1

    nen = get_nen()
    nes = get_nes()
    neq = get_neq()

    ien = get_ienarray()

    display_phi_plot = True

    #xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh vertices
    xpoints = get_xpoints()

    phi_sfem = np.zeros(Nn)

    # note 1D and linear elements, and ordered numbering leads to a tridiagonal banded matrix
    Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including all mesh vertices
    b = np.zeros(Nn) # right-hand-side vector including all mesh vertices

    # apply BCs
    phi_sfem[0] = get_left_bdry_value() # left BC
    phi_sfem[Nn-1] = get_right_bdry_value() # right BC

    xieq, weq = get_xieq_and_weq()
    shp, shpdlcl = get_shp_and_shpdlcl() # same type of elements in the entire mesh

    # loop over mesh cells
    for e in range(Ne): # loop index in [0,Ne-1]
        h = get_h(e)
        tau = get_tau(e) # constant over mesh when ax, kappa and h are constants
        kappa_num = tau*ax*ax # constant over mesh when tau and ax are constants

```

```
        # local/element-level data (matrix and vector)
        assert(nes==nen) # linear elements
        Ae = np.zeros([nen,nen])
        be = np.zeros(nen)

        jac = h/2.0 # 1D and linear elements with uniform spacing
        jacinv = 1/jac # 1D and linear elements
        detj = jac # 1D
        s = 1.0
        shpdgbl = jacinv*shpdlcl

        for q in range(neq): # loop index in [0,neq-1]
            wdetj = weq[q]*detj
            for idx_a in range(nes): # loop index in [0,nes-1]
                be[idx_a] = be[idx_a] + s*shp[idx_a,q]*wdetj +  s*ax*tau*shpdgbl[idx_a,q]*wdetj# no source term
                for idx_b in range(nes): # loop index in [0,nes-1]
                    Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
                                      - (shpdgbl[idx_a,q])*ax*shp[idx_b,q]*wdetj \
                                      + (shpdgbl[idx_a,q])*(kappa+kappa_num)*(shpdgbl[idx_b,q])*wdetj

        # assembly: recall 1D and linear elements, and ordered numbering for a tridiagonal matrix
        for idx_a in range(nes): # loop index in [0,nes-1]
            b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
            Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[idx_a,idx_a]
        Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper side of diagonal
        Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower side of diagonal

    # account for BCs in b
    # for now we assume Dirichlet BCs are zero (on left and right ends of the domain)
    b[0] = phi_sfem[0]
    b[1] = b[1] - Abanded[2,0]*b[0]
    b[Nn-1] = phi_sfem[Nn-1]
    b[Nn-2] = b[Nn-2] - Abanded[0,Nn-1]*b[Nn-1]
    Abanded[1,0] = 1.0
    Abanded[0,1] = 0.0 # upper side of diagonal
    Abanded[2,0] = 0.0 # lower side of diagonal
    Abanded[0,Nn-1] = 0.0 # upper side of diagonal
    Abanded[2,Nn-2] = 0.0 # lower side of diagonal
    Abanded[1,Nn-1] = 1.0

    phi_sfem = solve_banded((1,1),Abanded,b)

    if (display_phi_plot):
        plt.plot(xpoints,phi_sfem,'r*-')
        plt.xlabel('x')
        plt.ylabel('phi(x)')
        plt.title('phi(x) v x - Q2a')
        plt.savefig('Q2a.pdf')
        plt.show()

```

```
apply_num_scheme()
```

Listing 2: Python code for Q.2

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

```

(a) $M0$(uniform): Ne = 8 with node locations of: {0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0}

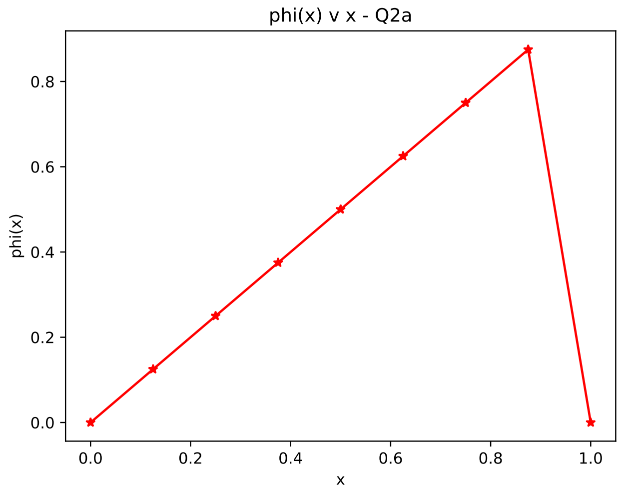
(b) $M1$(non-uniform): Ne = 4 with node locations of: {0.0,0.5,0.75,0.875,1.0}

(c) $M2$(non-uniform): Ne = 2 with node locations of: {0.0,0.875,1.0}

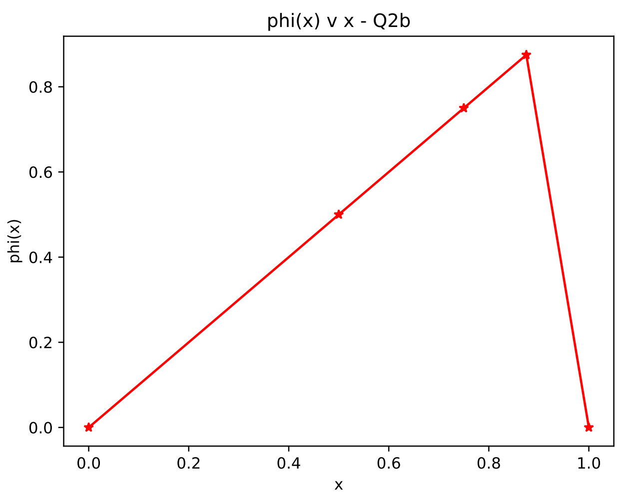
Solutions to $M0$, $M1$ and $M2$ mesh resolutions are in Fig 2.

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 Lisitng 3.

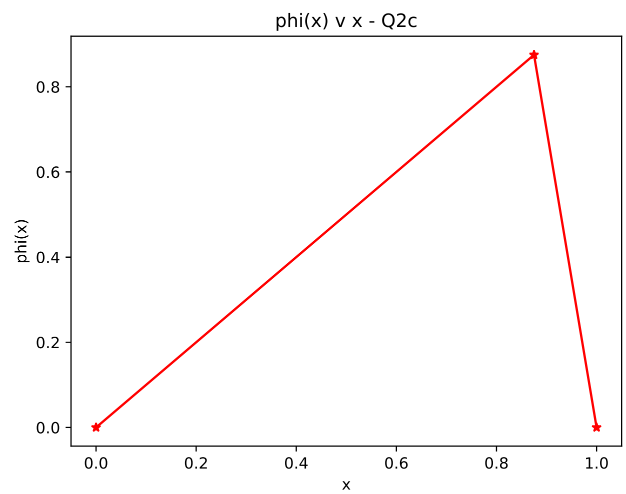
```
from webbrowser import get
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve_banded

def get_xmin():
    # return left end of domain
    xmin = 0.0
    return xmin

def get_xmax():
    # return right end of domain
    xmax = 1.0
    return xmax

def get_L():
    # return length of domain
    L = get_xmax()-get_xmin()
    return L

def get_ax():
    # return advection velocity value
    ax = 1.0e-0
    assert(np.abs(ax)>0)
    return ax

def get_kappa():
    # return kappa value
    kappa = 1.0e-1
    assert(kappa>0)
    return kappa

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

```python
    # return 'c' or reactivity (c<0 implies production and c>0 implies destruction)
    c = 1.0e+2
    ax = get_ax()
    kappa = get_kappa()
    assert(ax*ax+4*kappa*c>=0)
    return c

def get_Ne():
    # return number of elements in the mesh
    xp = get_xpoints()
    Ne = xp.size-1
    assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices for 2 domain end points)
    return Ne

def get_nen():
    # return number of vertices for an element
    nen = 2 # 1D
    return nen

def get_nes():
    # return number of shape/basis function for an element
    nes = 2 # 1D and linear
    return nes

def get_neq():
    # return number of numerical integration/quadrature points for an element
    neq = 2 # 2-point rule
    return neq

def get_xieq_and_weq():
    # return location of numerical integration/quadrature points in parent coordinates of an element
    neq = get_neq()
    assert(neq==2)
    xieq = np.zeros(neq)
    xieq[0] = -1.0/np.sqrt(3.0)
    xieq[1] =  1.0/np.sqrt(3.0)
    weq = np.zeros(neq)
    weq[0] = 1.0
    weq[1] = 1.0
    return xieq, weq

def get_xpoints():
    xpoints = np.array([0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0])
    # xpoints = np.array([0.0,0.125,0.25,0.5,0.75,0.875,1.0])
    return xpoints

def get_node(i):
    xpoints = get_xpoints()
    return xpoints[i]

def get_h(e):
    # return mesh size
    # h = get_L()/get_Ne() # uniform mesh
    h = get_node(e+1) - get_node(e)
    return h

```

```
def get_tau(e):
    # return tau alg1 value
    ax = get_ax()
    kappa = get_kappa()
    c = get_c()
    h = get_h(e)
    tau = 1.0/np.sqrt((2.0*ax/h)**2 + 9.0*(4.0*kappa/(h*h))**2+c*c)
    return tau

def get_ienarray():
    # return element-node connectivity
    Ne = get_Ne()
    nen = get_nen()
    ien = np.zeros([Ne,nen])
    # loop over mesh cells
    for e in range(Ne): # loop index in [0,Ne-1]
        ien[e,0] = e
        ien[e,1] = e+1
    return ien.astype(int)

def get_left_bdry_value():
    # return left bdry. value (Dirichlet BC)
    return 0.0

def get_right_bdry_value():
    # return right bdry. value (Dirichlet BC)
    return 1.0

def get_shp_and_shpdlcl():
    # return shape functions and derivatives evaluated at numerical integration/quadrature points
    nes = get_nes()
    neq = get_neq()
    xieq, weq = get_xieq_and_weq()
    assert(nes==2) # 1D and linear
    shp = np.zeros([nes,neq])
    shpdlcl = np.zeros([nes,neq]) # 1D
    for q in range(neq): # loop index in [0,neq-1]
        shp[0,q] = 0.5*(1-xieq[q])
        shpdlcl[0,q] = -0.5 # -1.0/2.0 for bi-unit 1D linear element
        shp[1,q] = 0.5*(1+ xieq[q])
        shpdlcl[1,q] = 0.5 # 1.0/2.0 for bi-unit 1D linear element
    return shp, shpdlcl

def apply_num_scheme():
    # apply numerical scheme

    xmin = get_xmin()
    xmax = get_xmax()

    ax = get_ax()
    kappa = get_kappa()
    c = get_c()

    Ne = get_Ne()
    Nn = Ne+1


    nen = get_nen()
```

```
    nes = get_nes()
    neq = get_neq()

    ien = get_ienarray()

    display_phi_plot = True

    # xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh vertices
    xpoints = get_xpoints()

    phi_sfem = np.zeros(Nn)

    # note 1D and linear elements, and ordered numbering leads to a tridiagonal banded matrix
    Abanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including all mesh vertices
    b = np.zeros(Nn) # right-hand-side vector including all mesh vertices

    # apply BCs
    phi_sfem[0] = get_left_bdry_value() # left BC
    phi_sfem[Nn-1] = get_right_bdry_value() # right BC

    xieq, weq = get_xieq_and_weq()
    shp, shpdlcl = get_shp_and_shpdlcl() # same type of elements in the entire mesh

    # loop over mesh cells
    for e in range(Ne): # loop index in [0,Ne-1]

        h = get_h(e)
        tau = get_tau(e) # constant over mesh when ax, kappa, c and h are constants
        kappa_num = tau*ax*ax # constant over mesh when tau and ax are constants
        ax_num1 = -tau*ax*c # constant over mesh when tau, ax and c are constants
        f_stab = 1.0 # f_stab=0.0 for SUPG, f_stab=-1.0 for GLS, or f_stab=1.0 for VMS
        ax_num2 = -f_stab*tau*c*ax # constant over mesh when tau, ax and c are constants
        c_num = -f_stab*tau*c*c # constant over mesh when tau and c are constants

        # local/element-level data (matrix and vector)
        assert(nes==nen) # linear elements
        Ae = np.zeros([nen,nen])
        be = np.zeros(nen)

        jac = h/2.0 # 1D and linear elements with uniform spacing
        jacinv = 1/jac # 1D and linear elements
        detj = jac # 1D
        s = 10.0
        shpdgbl = jacinv*shpdlcl

        for q in range(neq): # loop index in [0,neq-1]
            wdetj = weq[q]*detj
            for idx_a in range(nes): # loop index in [0,nes-1]
                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
                for idx_b in range(nes): # loop index in [0,nes-1]
                    Ae[idx_a,idx_b] = Ae[idx_a,idx_b] \
                                      - (shpdgbl[idx_a,q])*(ax+ax_num1)*shp[idx_b,q]*wdetj \
                                      + (shpdgbl[idx_a,q])*(kappa+kappa_num)*(shpdgbl[idx_b,q])*wdetj \
                                      + shp[idx_a,q]*ax_num2*(shpdgbl[idx_b,q])*wdetj \
                                      + shp[idx_a,q]*(c+c_num)*shp[idx_b,q]*wdetj

        # assembly: recall 1D and linear elements, and ordered numbering for a tridiagonal matrix
        for idx_a in range(nes): # loop index in [0,nes-1]
            b[ien[e,idx_a]] = b[ien[e,idx_a]] + be[idx_a]
            Abanded[1,ien[e,idx_a]] = Abanded[1,ien[e,idx_a]] + Ae[idx_a,idx_a]
        Abanded[0,ien[e,1]] = Abanded[0,ien[e,1]] + Ae[0,1] # upper side of diagonal
        Abanded[2,ien[e,0]] = Abanded[2,ien[e,0]] + Ae[1,0] # lower side of diagonal

    # account for BCs in b
    # for now we assume Dirichlet BCs are zero (on left and right ends of the domain)
    b[0] = phi_sfem[0]
    b[1] = b[1] - Abanded[2,0]*b[0]
    b[Nn-1] = phi_sfem[Nn-1]
    b[Nn-2] = b[Nn-2] - Abanded[0,Nn-1]*b[Nn-1]
    Abanded[1,0] = 1.0
    Abanded[0,1] = 0.0 # upper side of diagonal
    Abanded[2,0] = 0.0 # lower side of diagonal
    Abanded[0,Nn-1] = 0.0 # upper side of diagonal
    Abanded[2,Nn-2] = 0.0 # lower side of diagonal
    Abanded[1,Nn-1] = 1.0

    phi_sfem = solve_banded((1,1),Abanded,b)

    if (display_phi_plot):
        plt.plot(xpoints,phi_sfem,'r*-')
        plt.xlabel('x')
        plt.ylabel('phi(x)')
        plt.title('phi(x) v x - Q3a')
        plt.savefig('Q3a.pdf')
        plt.show()

apply_num_scheme()
```

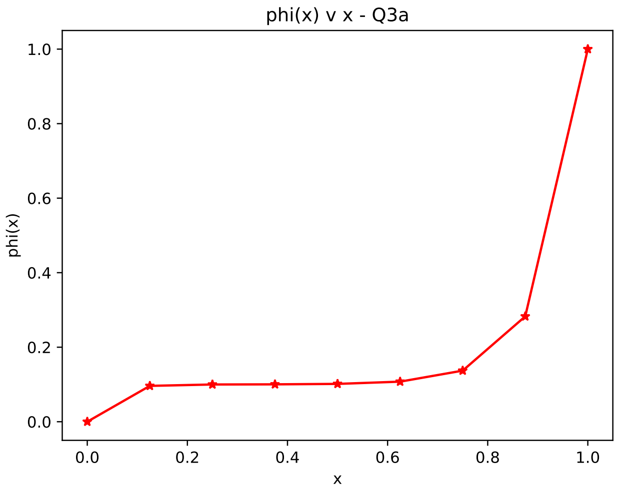
Listing 3: Python code for Q3.

```
        ax = 1.0
        kappa = 1.0e-1
        c = 1.0e+2
        s = 10.0
        phi(x = 0) = 0.0
        phi(x = L) = 1.0

```

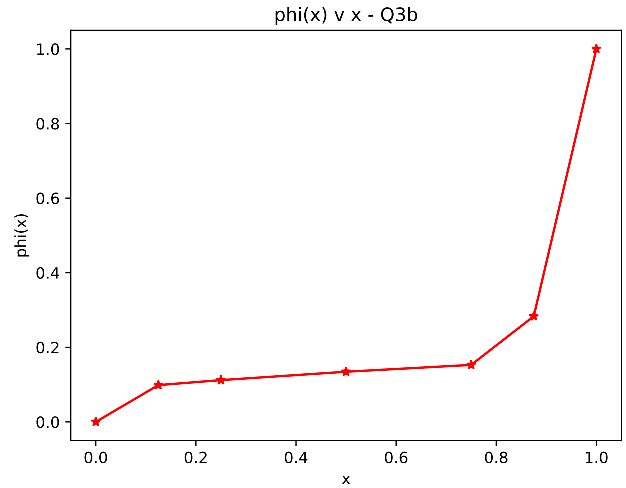
(a) $M0$(uniform): Ne $= 8$ with node locations of: $\{0.0,0.125,0.25,0.375,0.5,0.625,0.75,0.875,1.0\}$

(b) $M1$(non-uniform): Ne $= 6$ 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)