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

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., $t \in \left[0, T = L^2/\kappa = 40.0\right]$. Set BCs as: $\phi\left(x = 0, t\right) = \phi_0\left(t\right)$ and $\phi\left(x = L, t\right) = \phi_L\left(t\right) = g(t)$, where $g(t) = \min\left(1.25t/T, 1\right)$, 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.

```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_tmin():
    # return min time
    tmin = 0.0
    return tmin

def get_tmax():
    # return max time
    L = get_L()
    kappa = get_kappa()
    tmax = (L**2)/kappa
    return tmax

def get_T():
    # return the total time duration
    T = get_tmax()-get_tmin()
    return T

```

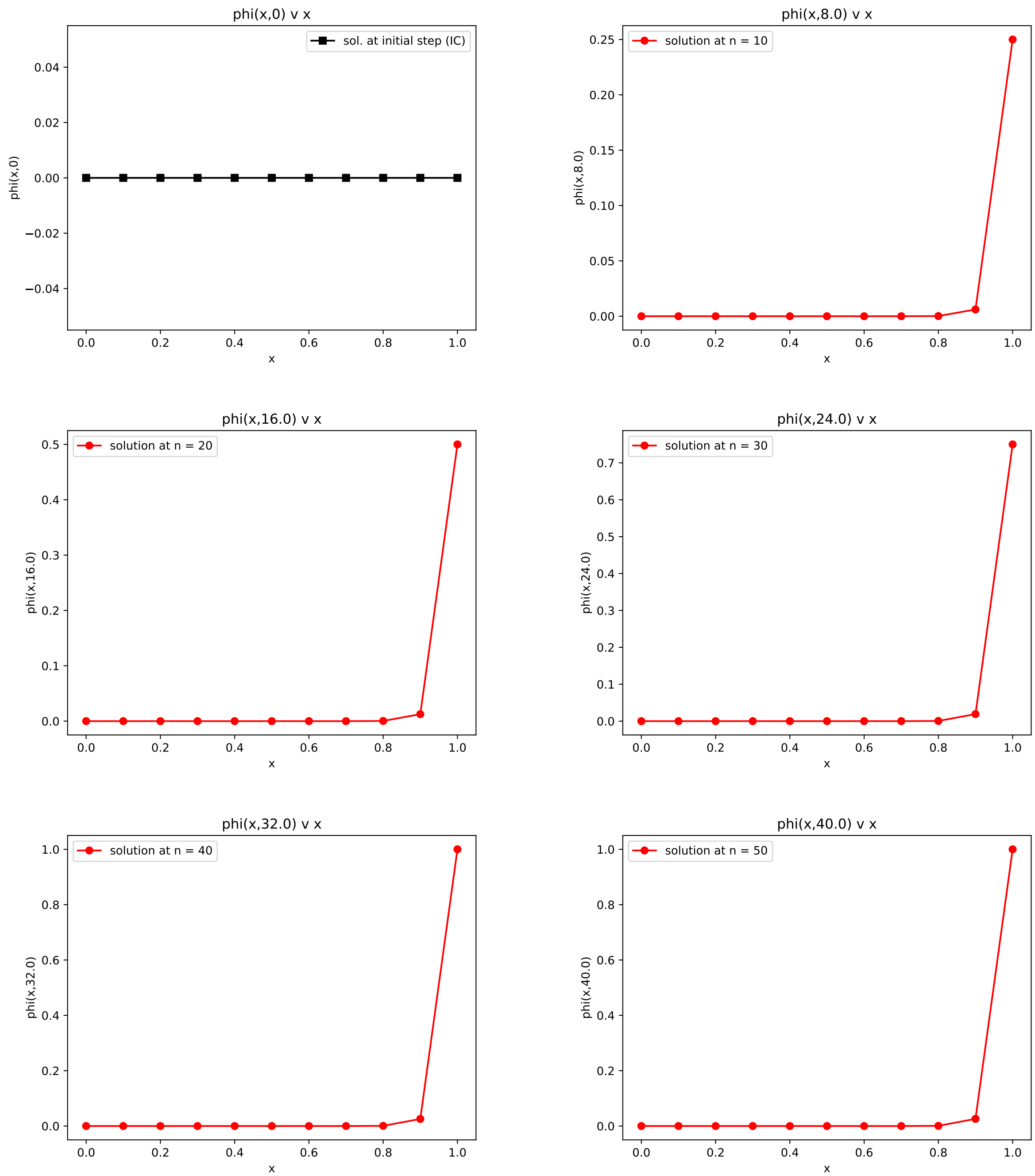


Figure 1: Solution $\phi(x,t)$ at various times

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

def get_kappa():
    # return kappa value
    kappa = 2.5e-2
    assert(kappa>0)
    return kappa

def get_Ne():
    # return number of elements in the mesh
    Ne = 10
    assert(Ne>1) # need more than 1 element (otherwise only 2 mesh vertices for 2 domain end points)
    return Ne

def get_Nt():
    # return number of time intervals
    Nt = 50 # note number of steps is Nt+1 including t_0 for IC
    return Nt

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 = 1 # 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

def get_h():
    # return mesh size
    h = get_L()/get_Ne() # uniform mesh
    return h

def get_dt():
    # return time-step size
    dt = get_T()/get_Nt() # uniform time intervals
    return dt
```

```
def get_tau():
    # return tau value
    ax = get_ax()
    kappa = get_kappa()
    h = get_h()
    dt = get_dt()
    tau = 1.0/np.sqrt((2.0/dt)**2 + (2.0*ax/h)**2 + 9.0*(4.0*kappa/(h*h))
    **2)
    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_IC():
    # return IC - 0 at all nodes at t=0
    Ne = get_Ne()
    Nn = Ne+1
    phi_sfem = np.zeros(Nn)
    return phi_sfem

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

def get_right_bdry_value(n):
    tmin = get_tmin()
    dt   = get_dt()
    T    = get_T()
    t    = tmin + n*dt
    val  = (1.25*t)/T
    if (val < 1.0):
        return val
    else:
        return 1.0

def get_source(x,n):
    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()

    tmin = get_tmin()
    tmax = get_tmax()

    ax = get_ax()
    kappa = get_kappa()

    Ne = get_Ne()
    Nn = Ne+1
    h = get_h()

    Nt = get_Nt()
    dt = get_dt()

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

    ien = get_ienarray()

    display_phi_plot = True

    tau = get_tau() # constant over mesh when ax, kappa and h are constants

    print('Pee:',0.5*np.abs(ax)*h/kappa) # debug
    print('CFL:,',np.abs(ax)*dt/h)
    print('tau:',tau) # debug

    kappa_num = tau*ax*ax # constant over mesh when tau and ax are constants

    xpoints = np.linspace(xmin,xmax,Nn,endpoint=True) # location of mesh vertices
    tpoints = np.linspace(tmin,tmax,Nt,endpoint=True) # location of time points

    # get IC (which satisfies BCs)
    phi_sfem = get_IC()
    if (display_phi_plot):
        plt.plot(xpoints,phi_sfem,'ks-',label='sol. at initial step (IC)')
        plt.xlabel('x')
        plt.ylabel('phi(x,0)')
        plt.legend(loc='upper right')
        plt.title('phi(x,0) v x')
        plt.savefig('Q1_IC.pdf')
        plt.show()
    # print(phi_sfem) # debug

```

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

    # loop over time intervals
    for n in range(1,Nt+1): # loop index [1,Nt]
        # note 1D and linear elements, and ordered numbering leads to a tridiagonal banded matrix
        Kbanded = np.zeros([3,Nn]) # left-hand-side (tridiagonal) matrix including all mesh vertices
        d = np.zeros(Nn) # right-hand-side vector including all mesh vertices

        # loop over mesh cells
        for e in range(Ne): # loop index in [0,Ne-1]
            # local/element-level data (matrix and vector)
            assert(nes==nen) # linear elements
            Ke = np.zeros([nen,nen])
            Me = np.zeros([nen,nen])
            Ae = np.zeros([nen,nen])
            be = np.zeros(nen)
            de = 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

            for q in range(neq): # loop index in [0,neq-1]
                wdetj = weq[q]*detj
                phiq = 0.0
                phidgblq = 0.0
                xq = 0.0

                for idx_a in range(nes): # loop index in [0,nes-1]
                    phiq = phiq + shp[idx_a,q]*phi_sfem[ien[e,idx_a]]
                    phidgblq = phidgblq + (shpdgbl[idx_a,q])*phi_sfem[ien[e,idx_a]]
                    xq = xq + shp[idx_a,q]*xpoints[ien[e,idx_a]]

                s = get_source(xq,n)
                for idx_a in range(nes): # loop index in [0,nes-1]
                    be[idx_a] = be[idx_a] + shp[idx_a,q]*s + shpdgbl[idx_a,q]*tau*ax*s

                    for idx_b in range(nes): # loop index in [0,nes-1]
                        Me[idx_a,idx_b] = Me[idx_a,idx_b] + shp[idx_a,q]*shp[idx_b,q]*wdetj \
                                                  + shpdgbl[idx_a,q]*(ax*tau)*shp[idx_b,q]*wdetj
                        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 # ... to be implemented ...
                        Ke[idx_a,idx_b] = Me[idx_a,idx_b] + dt*Ae[idx_a,idx_b]
                        de[idx_a] = de[idx_a] + Me[idx_a,idx_b]*phi_sfem[ien[e,idx_b]]
```

```


            # 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]
                d[ien[e,idx_a]] = d[ien[e,idx_a]] + de[idx_a]
                Kbanded[1,ien[e,idx_a]] = Kbanded[1,ien[e,idx_a]] + Ke[idx_a,idx_a]
            Kbanded[0,ien[e,1]] = Kbanded[0,ien[e,1]] + Ke[0,1] # upper side of diagonal
            Kbanded[2,ien[e,0]] = Kbanded[2,ien[e,0]] + Ke[1,0] # lower side of diagonal

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

        # account for BCs in d
        d[0] = phi_sfem[0]
        d[1] = d[1] - Kbanded[2,0]*d[0]
        d[Nn-1] = phi_sfem[Nn-1]
        d[Nn-2] = d[Nn-2] - Kbanded[0,Nn-1]*d[Nn-1]
        Kbanded[1,0] = 1.0
        Kbanded[0,1] = 0.0 # upper side of diagonal
        Kbanded[2,0] = 0.0 # lower side of diagonal
        Kbanded[0,Nn-1] = 0.0 # upper side of diagonal
        Kbanded[2,Nn-2] = 0.0 # lower side of diagonal
        Kbanded[1,Nn-1] = 1.0

        phi_sfem = solve_banded((1,1),Kbanded,d)

        if (n%10==0):
            s = 'solution at n = ' + str(n)
            s2 = 'Q1_n_'+str(n)+'.pdf'
            t_npo = get_tmin() + n*get_dt()
            s3 = 'phi(x,'+str(t_npo)+')'
            s4 = s3+' v x'
            if (display_phi_plot):
                plt.plot(xpoints,phi_sfem,'ro-',label=s)
                plt.legend(loc='upper left')
                plt.xlabel('x')
                plt.ylabel(s3)
                plt.title(s4)
                plt.savefig(s2)
                plt.show()


apply_num_scheme()
```

Listing 1: Transient Advection Diffusion - SUPG stabilized code

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

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

Use the VMS formulation for linear finite elements such that:

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

Set problem parameters as: $a_x = 1, \kappa = 1.0e-1, c = c_0\left(1.0+0.01\phi\right), c_0 = 1.0e+2$ and $s = 10.0$. Consider a domain length of $L = 1$, i.e., $x \in [0, L = 1]$. Set BCs as: $\phi\left(x=0\right) = \phi_0 = 0$ 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:

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

where,

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

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:

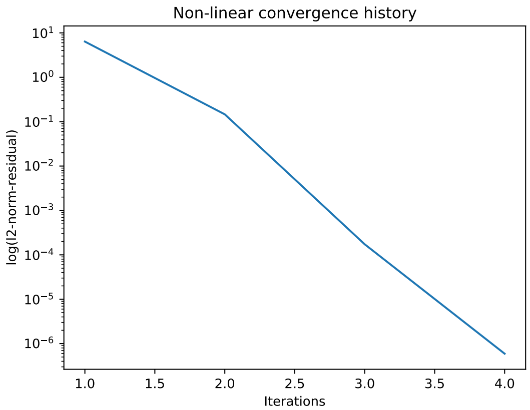
$$\begin{aligned}
G_a^e &= G_{a_l}^e + G_{a_{nl}}^e \\
G_{a_l}^e &= \left[\int_{x_l}^{x_r} \left(-N_{a,x}a_x\bar{\phi}\right) + \left(N_{a,x}\kappa\bar{\phi}_{,x}\right) + \left(-N_a s\right) + \left(N_{a,x}\tau a_x^2\bar{\phi}_{,x}\right) + \left(-N_{a,x}\tau a_x s\right)\,dx\right] \\
G_{a_{nl}}^e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

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

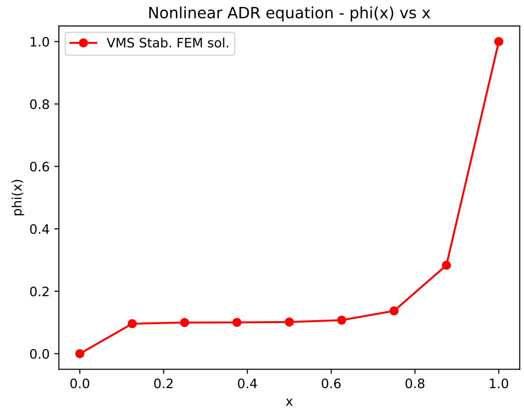
Expression for linearization is:

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

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



Non-linear convergence history



Solution plot

```
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): 0.022579528620555508
NL iter (starting at 0): 2
l2 norm of non-linear weak residual: 0.0001729513071683056
max. nodal value of update (abs. value): 4.409347702675172 4e-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

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

(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

```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-1
    assert(kappa > 0.0)
    return kappa

def get_c0():
    # return c0 value
    c0 = 1.0e2
    return c0

def get_c(phi):
    # return c value
    c0 = get_c0()
    c  = c0*(1.0 + 0.01*phi)
    ax = get_ax()
    kappa = get_kappa()
    assert(ax*ax+4*kappa*c>=0)
    return c

def get_dc_dphi(phi):
    # return dc_dphi value: \frac{\partial c}{\partial \phi}
    dcdphi = (get_c0())*0.01
    return dcdphi

def get_source():
    # return s value
    s = 10.0
    return s

def get_Ne():
    # return number of elements in the mesh
    Ne = 8
    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
    # integrates upto 9th order polynomial accurately
    neq = 5 # 5-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==5)
    xieq = np.zeros(neq)
    xieq[0] = (-1.0/3.0)*np.sqrt( 5.0 + 2.0*np.sqrt( 10.0/7.0 ) )
    xieq[1] = (-1.0/3.0)*np.sqrt( 5.0 - 2.0*np.sqrt( 10.0/7.0 ) )
    xieq[2] = 0.0
    xieq[3] = (1.0/3.0)*np.sqrt( 5.0 - 2.0*np.sqrt( 10.0/7.0 ) )
    xieq[4] = (1.0/3.0)*np.sqrt( 5.0 + 2.0*np.sqrt( 10.0/7.0 ) )
    weq = np.zeros(neq)
    weq[0] = ( 322.0 - 13.0*np.sqrt(70.0) )/900.0
    weq[1] = ( 322.0 + 13.0*np.sqrt(70.0) )/900.0
    weq[2] = 128.0/225.0
    weq[3] = ( 322.0 + 13.0*np.sqrt(70.0) )/900.0
    weq[4] = ( 322.0 - 13.0*np.sqrt(70.0) )/900.0
    return xieq, weq

def get_h():
    # return mesh size
    h = get_L()/get_Ne() # uniform mesh
    return h

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

    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()
    s = get_source() # constant source term

    Ne = get_Ne()
    Nn = Ne+1
    h = get_h()

    nen = get_nen()
    nes = get_nes()
    neq = get_neq()
    assert(nes==nen) # linear elements

    ien = get_ienarray()

    display_phi_plot = True

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

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

    phi_sfem = np.ones(Nn) # initial guess
    # apply Dirichlet/essential BCs to initial guess
    phi_sfem[0] = get_left_bdry_value() # left BC
    phi_sfem[Nn-1] = get_right_bdry_value() # right BC

    NLmaxiters = 100 # num. of NL max iterations
```

```
    NLtol = 1.0e-6 # tol. for NL weak residual
    max_abs_phi_del_tol = 1.0e-6 # tol. for max of absolute value of phi del/update
    norm_val1 = []
    norm_val2 = []
    iter = []
    converged_flag = 0
    # loop over NL iterations
    for k in range(NLmaxiters): # loop index in [0,NLmaxiters-1]

        iter.append(k+1)
        # 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

        # loop over mesh cells
        for e in range(Ne): # loop index in [0,Ne-1]
            # local/element-level data (matrix and vector)

            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

            for q in range(neq): # loop index in [0,neq-1]
                wdetj = weq[q]*detj

                phiq = 0.0
                phidgblq = 0.0

                for idx_a in range(nes): # loop index in [0,nes-1]
                    phiq = phiq + shp[idx_a,q]*phi_sfem[ien[e,idx_a]]
                    phidgblq = phidgblq + (shpdgbl[idx_a,q])*phi_sfem[ien[e,idx_a]]

                cq = get_c(phiq)
                dcdphiq = get_dc_dphi(phiq)
                tauq = get_tau(cq)
                kappa_numq = tauq*ax*ax
                for idx_a in range(nes): # loop index in [0,nes-1]
                    be[idx_a] = be[idx_a] \
                                - (shpdgbl[idx_a,q])*ax*phiq*wdetj \
                                + (shpdgbl[idx_a,q])*(kappa+kappa_numq)*(phidgblq)*wdetj \
                                - (shp[idx_a,q])*s*wdetj \
                                - (shpdgbl[idx_a,q])*tauq*ax*s*wdetj \
                                + (shp[idx_a,q])*cq*phiq*wdetj \
                                + (shpdgbl[idx_a,q])*tauq*ax*cq*phiq*wdetj \
                                - (shp[idx_a,q])*tauq*ax*cq*phidgblq*wdetj \
                                - (shp[idx_a,q])*tauq*cq*cq*phiq*wdetj \
```

```
                            + (shp[idx_a,q])*tauq*cq*s*wdetj

                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_numq)*(shpdgbl[idx_b,q])*wdetj \
                                      + (shp[idx_a,q])*(phiq*dcdphiq + cq)*(shp[idx_b,q])*wdetj \
                                      + (shpdgbl[idx_a,q])*tauq*ax*(phiq*dcdphiq + cq)*(shp[idx_b,q])*wdetj \
                                      - ((shp[idx_a,q])*tauq*ax*(phidgblq*dcdphiq)*(shp[idx_b,q]) + (shp[idx_a,q])*tauq*ax*cq*(shpdgbl[idx_b,q]))*wdetj \
                                      - (shp[idx_a,q])*tauq*(2.0*phiq*cq*dcdphiq + cq*cq)*(shp[idx_b,q])*wdetj \
                                      + (shp[idx_a,q])*tauq*s*dcdphiq*(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] = 0.0
    b[Nn-1] = 0.0
    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

    print('NL iter (starting at 0):',k)
    NLweak_res_l2 = np.linalg.norm(b)
    norm_val1.append(NLweak_res_l2)
    print('l2 norm of non-linear weak residual:',NLweak_res_l2)
    if (NLweak_res_l2<=NLtol):
       converged_flag = 1
       print('Converged (for NL weak residual')
       break;

    phi_sfem_del = solve_banded((1,1),Abanded,-b) # note minus sign with 'b' as in '-b'

    max_abs_phi_sfem_del = np.max(np.abs(phi_sfem_del))
    norm_val2.append(max_abs_phi_sfem_del)
    print('max. nodal value of update (abs. value):',
```

```
    max_abs_phi_sfem_del) # debug
        if (max_abs_phi_sfem_del <= max_abs_phi_del_tol):
            converged_flag = 2
            print('Converged (for update)')
            break;

        # apply update
        phi_sfem = phi_sfem + phi_sfem_del

    if (display_phi_plot):
        plt.plot(xpoints,phi_sfem,'ro-',label='VMS Stab. FEM sol.')
        plt.legend(loc='upper left')
        plt.xlabel('x')
        plt.ylabel('phi(x)')
        plt.title('Nonlinear ADR equation - phi(x) vs x')
        plt.savefig('Q2.pdf')
        plt.show()

    plt.semilogy(iter,norm_val1)
    plt.xlabel('Iterations')
    plt.ylabel('log(l2-norm-residual)')
    plt.title('Non-linear convergence history')
    plt.savefig('Q2-convergence.pdf')
    plt.show()

apply_num_scheme()
```

Listing 2: Steady, Non-linear ADR equation