MATH 352 - Spring 2021
Homework 5

Dr. Alessandro Veneziani

Kai Chang, Diego Gonzalez, Rodrigo Gonzalez, Kevin Hong, Katie Roebuck

---

1 Weak formulation for u(x)

Let $y(x) = \mu u'(x) - \psi'(x)u(x)$. The DE reads

$$-\frac{\mathrm{d}}{\mathrm{d}x}(y(x)) = 1$$

Multiplying by v and integrate over the domain to obtain

$$-\int_0^1 v(x)y'(x)dx = \int_0^1 v(x)dx$$

Integrating by parts, this is the same as

$$v(0)y(0) - v(1)y(1) + \int_0^1 v'(x)y(x)dx = \int_0^1 v(x)dx$$

Define the functional space as

$$V = \{z \in H^1(0,1) : z(0) = z(1) = 0\} \tag{1}$$

and let $u, v \in V$. Note that the $v(0)y(0)$ and $v(1)y(1)$ terms vanish. We have

$$\int_0^1 v'(x)y(x)dx = \int_0^1 v(x)dx$$

Substituting for $y(x)$, we obtain

$$\mu \int_0^1 u'(x)v'(x)dx - \int_0^1 \psi'(x)u(x)v'(x)dx = \int_0^1 v(x)dx$$

We may transfer the derivative from $v'(x)$ in the second integral by using integration by parts again to obtain

$$\mu \int_0^1 u'(x)v'(x)dx + \int_0^1 (\psi'(x)u(x))'v(x)dx = \int_0^1 v(x)dx \tag{2}$$

So we define the bilinear form as

$$a(u,v) = \mu \int_0^1 u'(x)v'(x)dx + \int_0^1 (\psi'(x)u(x))'v(x)dx \tag{3}$$

And the functional as

$$F(v) = \int_0^1 v(x)dx \tag{4}$$

So the weak formulation of the problem is:

Find $u \in V$ such that $a(u,v) = F(v)$, for any $v \in V$, where V, a, and F are defined as in (1), (3), and (4), respectively.

Note that if we assume $\psi(x) = \alpha x$, with α a constant, the bilinear form further simplifies to

$$a(u,v) = \mu \int_0^1 u'(x)v'(x)dx + \alpha \int_0^1 u'(x)v(x)dx \tag{5}$$

2 Weak formulation after change of variable

Given that

$$\frac{\mathrm{d}}{\mathrm{d}x}(u(x)) = \rho' e^{\psi/\mu} + \rho e^{\psi/\mu}(\psi'/\mu)$$

The DE reads

$$-\mu \frac{\mathrm{d}}{\mathrm{d}x}\left(\rho'(x) e^{\psi(x)/\mu}\right) = 1$$

which is the same as

$$\mu\rho''(x) + \psi'(x)\rho'(x) = -e^{-\psi(x)/\mu}$$

Multiplying by v, integrating over the domain, and using integration by parts, we obtain

$$\mu\left(\rho' v\Big|_0^1 - \int_0^1 \rho' v' dx\right) + \int_0^1 \psi'\rho' v dx = -\int_0^1 e^{-\psi/\mu} v dx$$

Note that ρ must satisfy the boundary conditions. We define the functional space as

$$V = \{z \in H^1(0,1) : z(0) = z(1) = 0\} \tag{6}$$

and let $\rho, v \in V$. Note that the $\rho' v\Big|_0^1$ term vanishes. We have

$$\mu \int_0^1 \rho' v' dx - \int_0^1 \psi'\rho' v dx = \int_0^1 e^{-\psi/\mu} v dx \tag{7}$$

So we define the bilinear form as

$$a(\rho, v) = \mu \int_0^1 \rho' v' dx - \int_0^1 \psi'\rho' v dx \tag{8}$$

And the functional as

$$F(v) = \int_0^1 e^{-\psi/\mu} v dx \tag{9}$$

So the weak formulation of the problem is:

Find $\rho \in V$ such that $a(\rho, v) = F(v)$, for any $v \in V$, where V, a, and F are defined as in (6), (8), and (9), respectively.

Note that if we assume $\psi(x) = \alpha x$, with α a constant, the bilinear form and the functional further simplify to

$$a(\rho, v) = \mu \int_0^1 \rho' v' dx - \alpha \int_0^1 \rho' v dx$$

$$F(v) = \int_0^1 e^{-\alpha x/\mu} v dx$$

3 Finite Element Approximation

Using finite elements, say in $\mathbb{P}^n$, we write the approximate solution u_h as

$$u_h = \sum_{i=0}^{N} u_i \phi_i \,.$$

Substituting u with u_h and v with ϕ_j in equation (2), we obtain the finite element approximation

$$\mu \sum_{i=0}^{N} u_i \int_0^1 \phi_i' \phi_j' + \sum_{i=0}^{N} u_i \int_0^1 (\psi' \phi_i)' \phi_j = \int_0^1 \phi_j$$

where j is ranging from 0 to N.

In particular, if we assume $\psi(x) = \alpha x$, it yields

$$\mu \sum_{i=0}^{N} u_i \int_0^1 \phi_i' \phi_j' + \alpha \sum_{i=0}^{N} u_i \int_0^1 \phi_i' \phi_j = \int_0^1 \phi_j \,.$$

This can be rewritten as a linear system of $N+1$ equations with the form of

$$Au = b.$$

Similarly, for the case after the change of variable, we write the approximate solution ρ_h as

$$\rho_h = \sum_{i=0}^{N} \rho_i \phi_i \,.$$

Substituting ρ with ρ_h and v with ϕ_j in equation (7), we obtain the finite element approximation

$$\mu \sum_{i=0}^{N} \rho_i \int_0^1 \phi_i' \phi_j' - \sum_{i=0}^{N} \rho_i \int_0^1 \psi' \phi_i' \phi_j = \int_0^1 e^{-\psi/\mu} \phi_i$$

where j is ranging from 0 to N.

If we assume $\psi(x) = \alpha x$, it yields

$$\mu \sum_{i=0}^{N} \rho_i \int_0^1 \phi_i' \phi_j' - \alpha \sum_{i=0}^{N} \rho_i \int_0^1 \phi_i' \phi_j = \int_0^1 e^{-\alpha x/\mu} \phi_j$$

This can be written as a linear system of $N+1$ equations with the form of

$$B\rho = c \,.$$

4 Verify the exact solution

Given $u(x)$ and $\psi(x)$, we have

$$\frac{\mathrm{d}}{\mathrm{d}x}u(x) = \frac{1}{\alpha}\left(1 - \frac{\alpha e^{\alpha x/\mu}}{\mu(e^{\alpha/\mu}-1)}\right)$$

which means that

$$\mu u' - \psi' u = \frac{\mu}{\alpha}\left(1 - \frac{\alpha e^{\alpha x/\mu}}{\mu(e^{\alpha/\mu}-1)}\right) - \frac{\alpha}{\alpha}\left(x - \frac{e^{\alpha x/\mu}-1}{(e^{\alpha/\mu}-1)}\right)$$

then

$$\mu u' - \psi' u = \frac{\mu}{\alpha} - \frac{1}{e^{\alpha/\mu}-1} - x$$

and

$$-\frac{\mathrm{d}}{\mathrm{d}x}(\mu u' - \psi' u) = 1$$

Note that

$$u(0) = \frac{1}{\alpha}\left(0 - \frac{1-1}{e^{\alpha/\mu}-1}\right) = 0$$

$$u(1) = \frac{1}{\alpha}\left(1 - \frac{e^{\alpha/\mu}-1}{e^{\alpha/\mu}-1}\right) = 0$$

So $u(x)$ solves the problem.
As for $\rho(x)$, we know it must satisfy the following DE

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(\mu\rho' e^{\alpha x/\mu}\right) = 1$$

So we compute the first derivative of ρ

$$\frac{\mathrm{d}}{\mathrm{d}x}\rho(x) = e^{-\alpha x/\mu}\left(-\frac{x}{\mu} + \frac{1}{\mu(1-e^{\alpha/mu})} + 1\right)$$

then

$$\mu\rho' e^{\alpha x/\mu} = -x + \frac{1}{1-e^{\alpha/\mu}} + \mu$$

Therefore, the DE is satisfied. As for the boundary conditions, notice

$$\rho(0) = \frac{1}{\alpha}\left(\frac{1-1}{1-e^{\alpha/\mu}} + 0\right) = 0$$

$$\rho(1) = \frac{1}{\alpha}\left(\frac{1-e^{-\alpha/\mu}}{1-e^{\alpha/\mu}} + e^{-\alpha/\mu}\right) = \frac{1}{\alpha}\left(\frac{1-e^{-\alpha/\mu}+e^{-\alpha/\mu}-1}{1-e^{\alpha/\mu}}\right) = 0$$

So $\rho(x)$ solves the problem.

5 Test for $u(x)$ with $\alpha = 1$

To numerically solve this problem, we adapted `fem1DP1.py` (you can see the exact code that we used on the Appendix). We computed the numerical solutions for this problem using values of $\alpha = 1$, $\mu = 0.1, 0.01$ and $h = 0.1, 0.01$. The results can be seen in Figures 1 and 2. For this set of values, the only unstable solution occurred with $h = 0.1$, $\mu = 0.01$. To solve this, we used an upwind scheme as suggested. Figure 2 shows the numerical solution using the upwind scheme in that case.

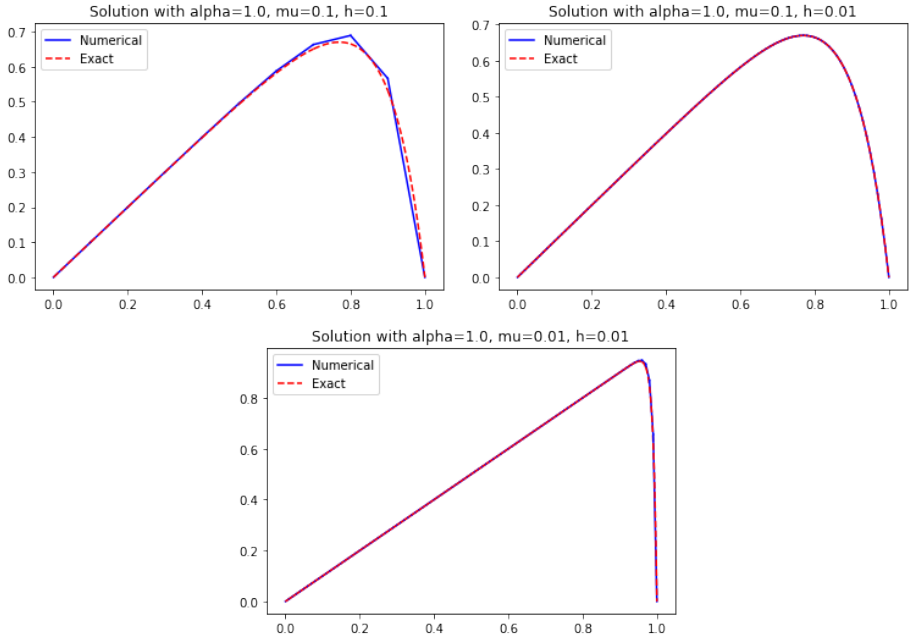


Figure 1: Numerical Solutions and their respective errors using different values for μ and h.

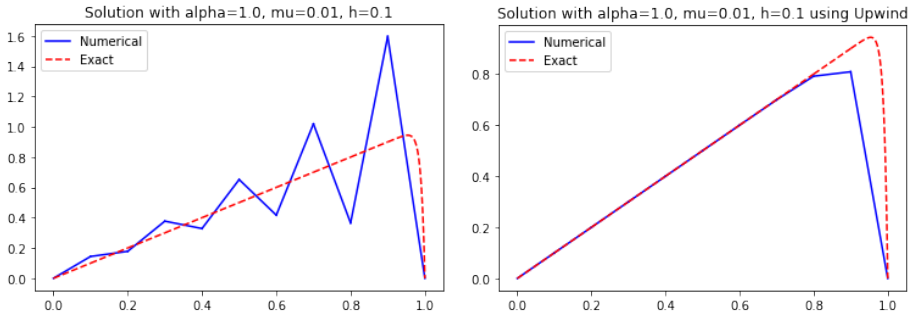


Figure 2: Numerical Solutions using values of $\mu = 0.01$ and $h = 0.1$. The first plot shows oscillations caused by an unstable numerical solution. This problem is solved by using an upwind scheme. As we can see on the second plot, the oscillations have disappeared.

6 Test for $u(x)$ with $\alpha = -1$

The next step was to use a value of $\alpha = -1$. The solutions behave as expected, as you can see in Figure 3. In this case, we again have stable solutions with the exception of $h = 0.1$, $\mu = 0.01$. In this case, using an upwind scheme eliminated all the oscillations of the unstable solution.

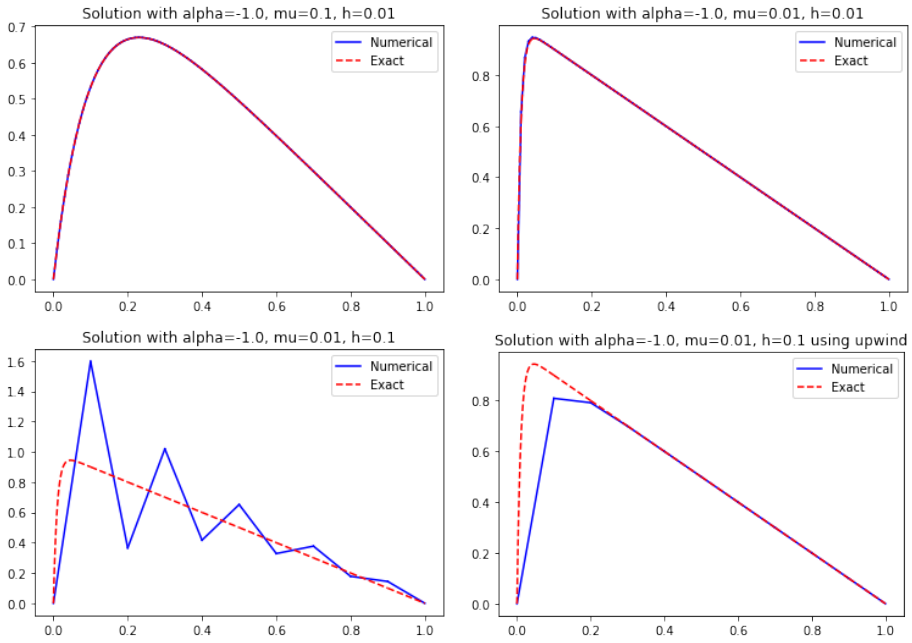


Figure 3: Numerical Solutions using different values for μ and h. The first row shows stable solutions, while in the second row we first see an unstable solution and then an application of the upwind scheme that makes it a stable solution.

7 Test for $\rho(x)$ with $\alpha = 1$

To numerically solve this problem, we again adapted `fem1DP1.py` (you can also see the exact code that we used on the Appendix). We computed the numerical solutions for this problem using values of $\alpha = 1$, $\mu = 0.1, 0.01$ and $h = 0.1, 0.01$. To retrieve u from ρ, we simply used the suggested change of variable. The results can be seen in Figures 4 and 5. For this set of values, the only unstable solution occurred with $h = 0.1$, $\mu = 0.01$. To solve this, we used an upwind scheme as suggested. Figure 5 shows the numerical solution using the upwind scheme in that case.

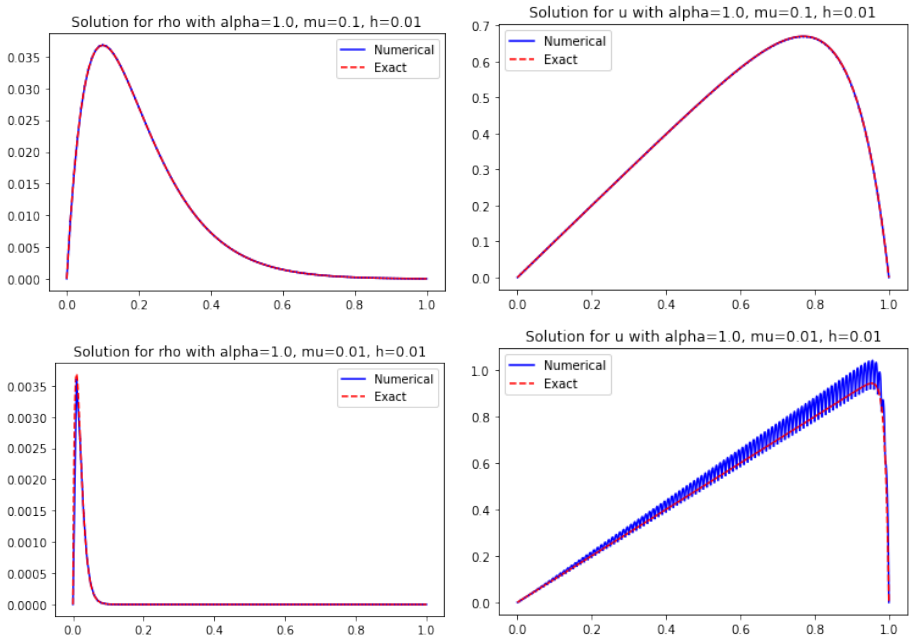


Figure 4: Numerical Solutions using different values for μ and h. The column shows the numerical solutions for ρ, while the second column shows the respective u obtained from these.

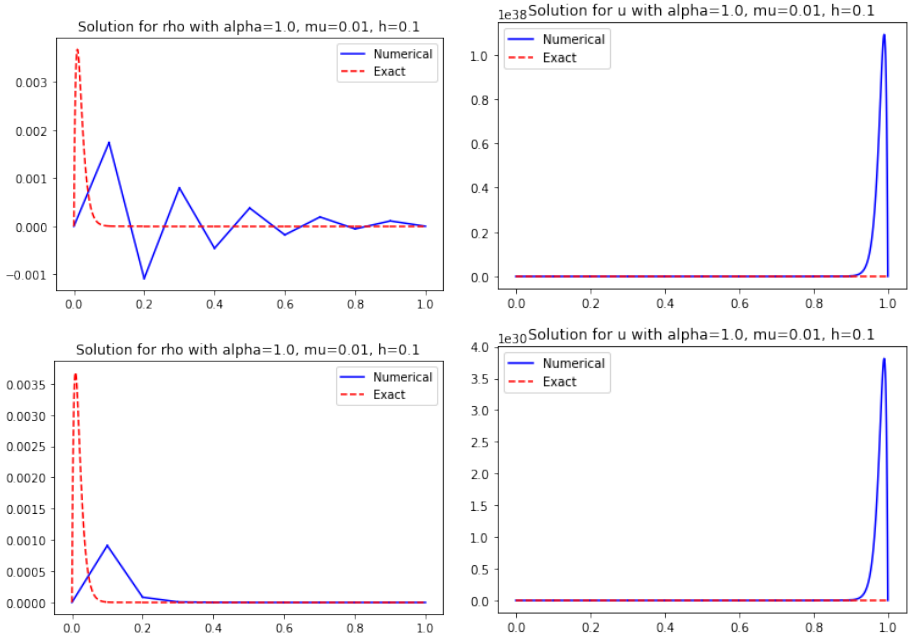


Figure 5: Numerical Solutions using values of $\mu = 0.01$ and $h = 0.1$. The first row shows oscillations on ρ, and the retrieved u is a bad numerical solution as expected. The second row shows the same problem solved with an upwind scheme. As we can appreciate, the oscillations dissappear for ρ, but still the solution of u does not adapt well. This is because when we solve for the original problem (i.e. when we solve for u), the stabilization is made with respect to u, which is why we can expect the oscillation for u disappeared. But when we use the change of variable formula, the stabilization is made with respect to ρ, which does not naturally adapt to u (as one may notice the sign of α actually changed after change of variable which changes the nature of the problem). Consequently, u in this case is also a bad approximation.

8 Test for $\rho(x)$ with $\alpha = -1$

Lastly, we tested this problem using a value of $\alpha = -1$. Here again, we see the same behavior for the numerical solutions that we found in the previous section.

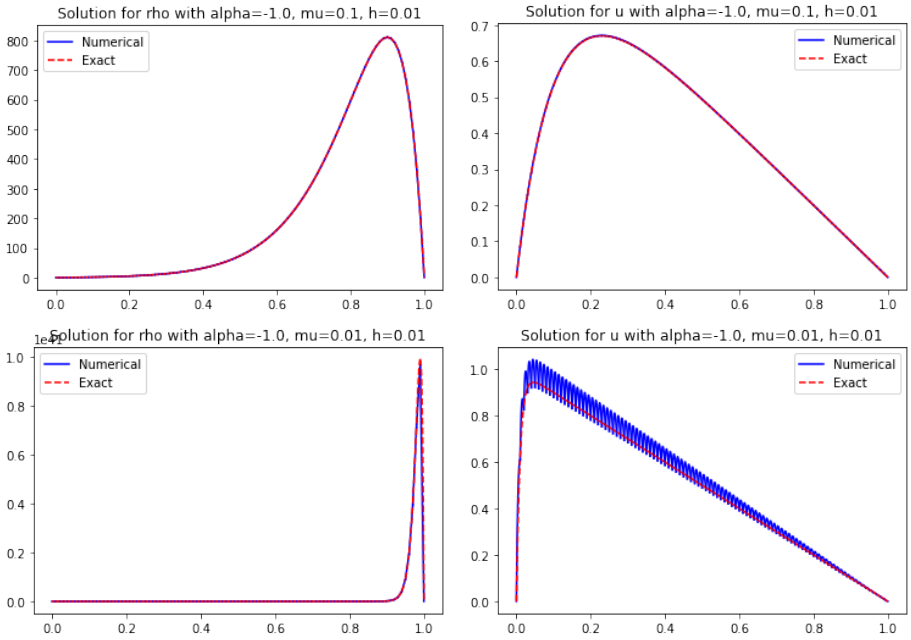


Figure 6: Numerical Solutions using different values for μ and h. The column shows the numerical solutions for ρ, while the second column shows the respective u obtained from these.

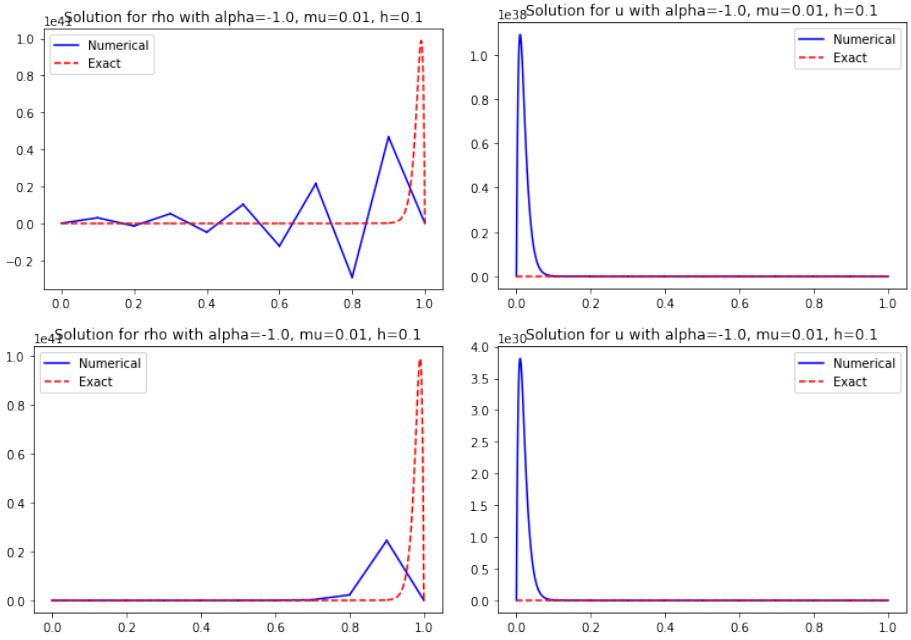


Figure 7: Numerical Solutions using values of $\mu = 0.01$ and $h = 0.1$. The first row shows oscillations on ρ, and the retrieved u is a bad numerical solution as expected. The second row shows the same problem solved with an upwind scheme. As we can appreciate, the oscillations dissappear for ρ, but the solution of u still does not adapt well. This is because when we solve for the original problem (i.e. when we solve for u), the stabilization is made with respect to u, which is why we can expect the oscillation for u disappeared. But when we use the change of variable formula, the stabilization is made with respect to ρ, which does not naturally adapt to u (as one may notice the sign of α actually changed after change of variable which changes the nature of the problem). Consequently, u in this case is also a bad approximation.

9 Weak formulation for diffusion-advection problem

Define the functional space as

$$V = \{z \in H^1(\Omega) : z(\partial\Omega) = 0\} \tag{10}$$

Multiply by $v \in V$, and integrate over the domain to obtain

$$-\mu \int_\Omega (\vec{\nabla} \cdot \vec{\nabla} c)v + \int_\Omega (\vec{\beta} \cdot \vec{\nabla} c)v = 0$$

By Green's formula (integration by parts), we have

$$-\mu \int_\Omega (\vec{\nabla} \cdot \vec{\nabla} c)v = -\mu \left(\int_{\partial\Omega} (\vec{\nabla} c \cdot \hat{n})v - \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c \right)$$

The term integrating over the boundary disappears since v vanishes there. Therefore, the equation reads

$$\mu \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \int_\Omega (\vec{\beta} \cdot \vec{\nabla} c)v = 0$$

So we define the bilinear form as

$$a(c, v) = \mu \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \int_\Omega (\vec{\beta} \cdot \vec{\nabla} c)v \tag{11}$$

And the functional as

$$F(v) = 0 \tag{12}$$

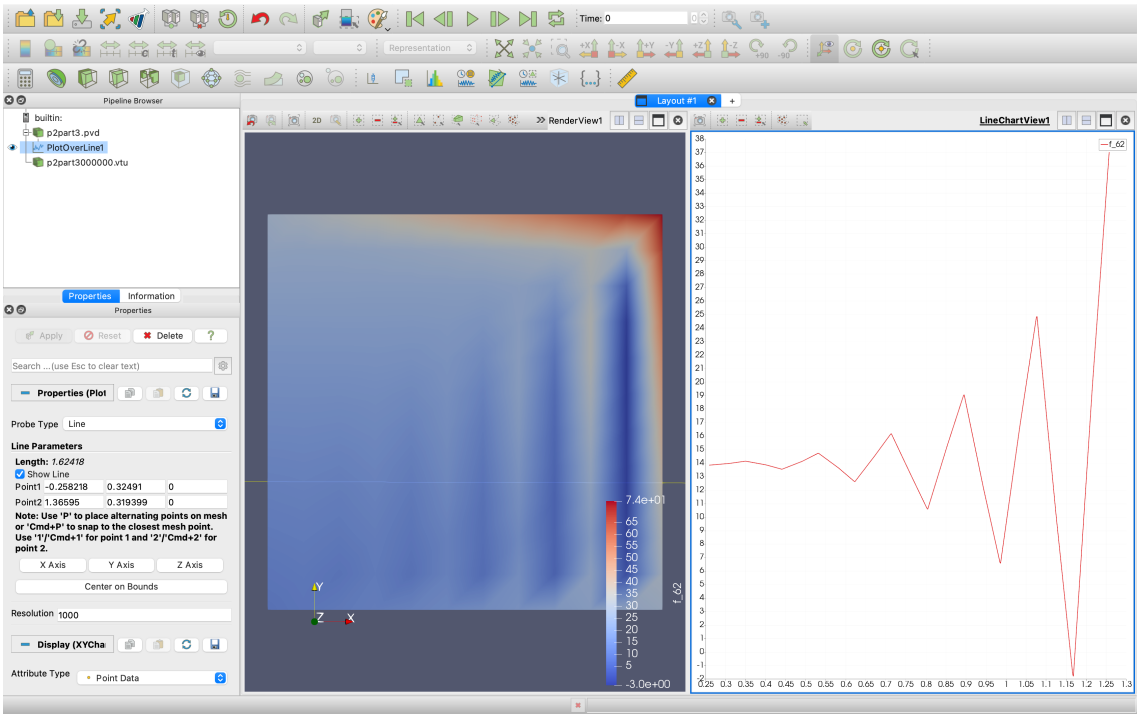
So the weak formulation of the problem is:

Find $c \in V \bigoplus \mathcal{L}$ such that $a(c, v) = F(v)$, for any $v \in V$, where V, a, and F are defined as in (7), (8), and (9), respectively.

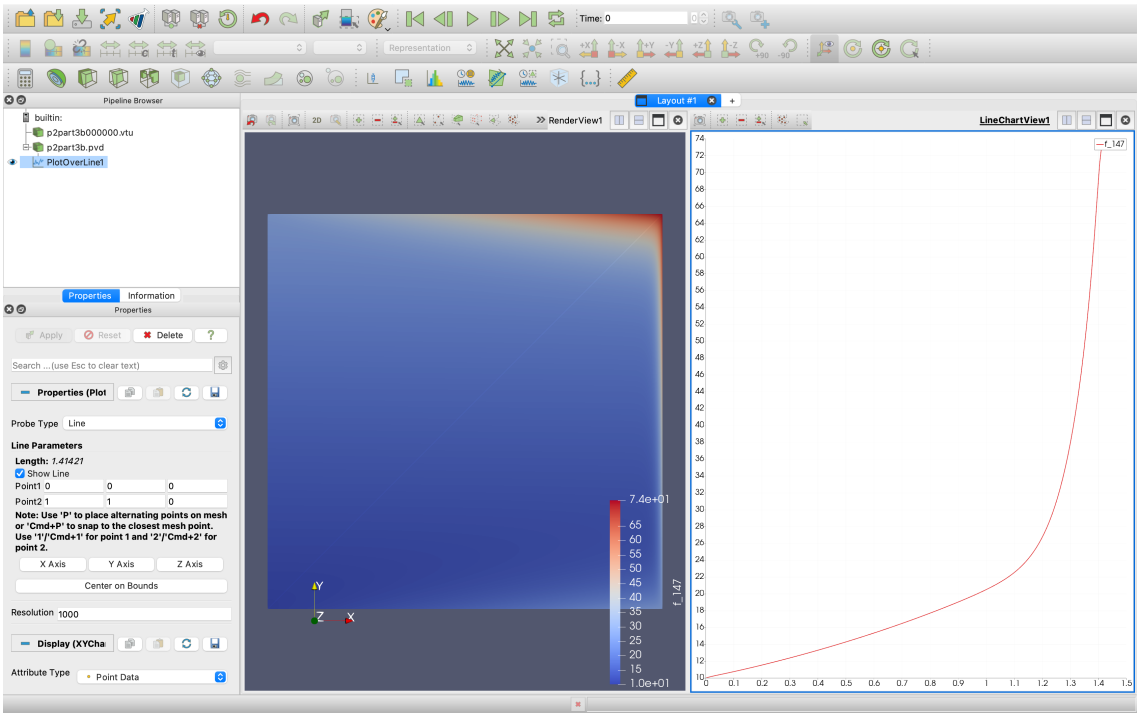
Note that c must be the sum of an element of V and a lifting function.

10 Test with 0.1 diffusivity

Using $\mu = 0.1$ and $h \approx 0.129$ (11 elements per side, the general code used for this problem can be found in the Appendix) gives $\mathbb{P}\text{e} = |\vec{\beta}|h/(2\mu) \approx 6.428 > 1$, which implies that we will have oscillations. Indeed, after running the code, we obtain the following graph with clear oscillations.

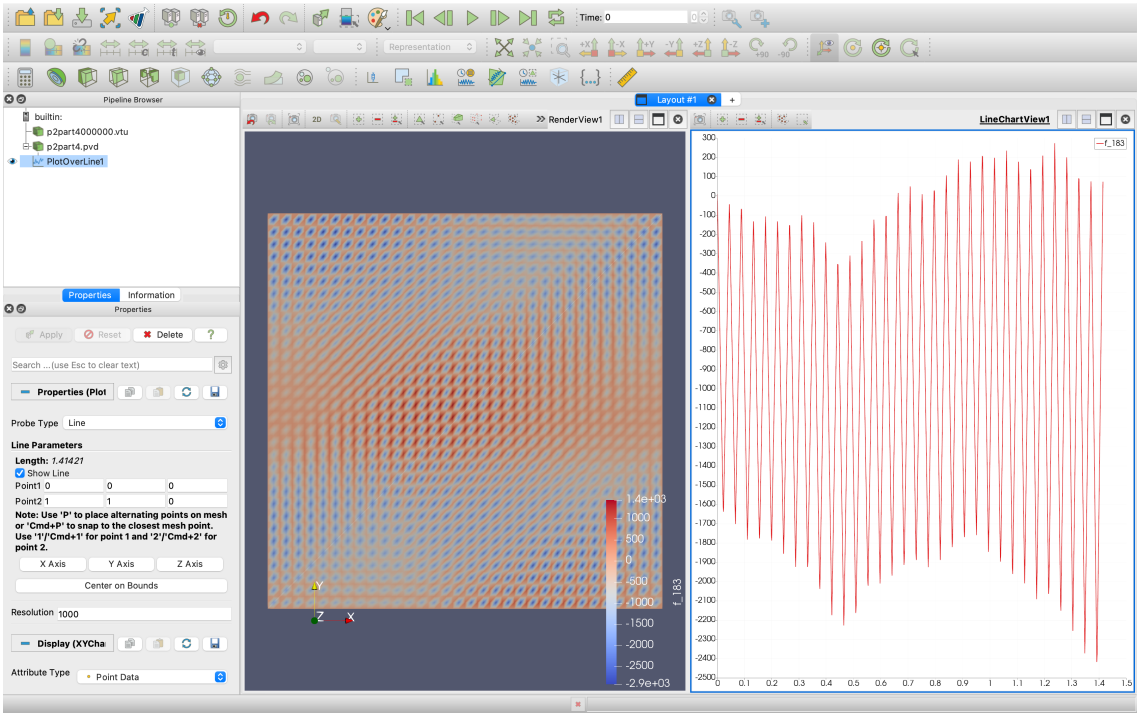


Using $\mu = 0.1$, we now adjust the space reticulation with $h \approx 0.009$ (150 elements per side) which means that $\mathbb{P}\mathrm{e} \approx 0.471 < 1$, which implies that we will not have oscillations. Indeed, after running the code we obtain the following graph without oscillations.



11 Test with 0.0001 diffusivity

Using $\mu = 0.0001$ and $h = ...$ (64 elements per side) gives $\mathbb{P}e = ... > 1$, which implies that we will have oscillations. Indeed, after running the code we obtain the following graph with unacceptable oscillations.



12 Artificial viscosity method

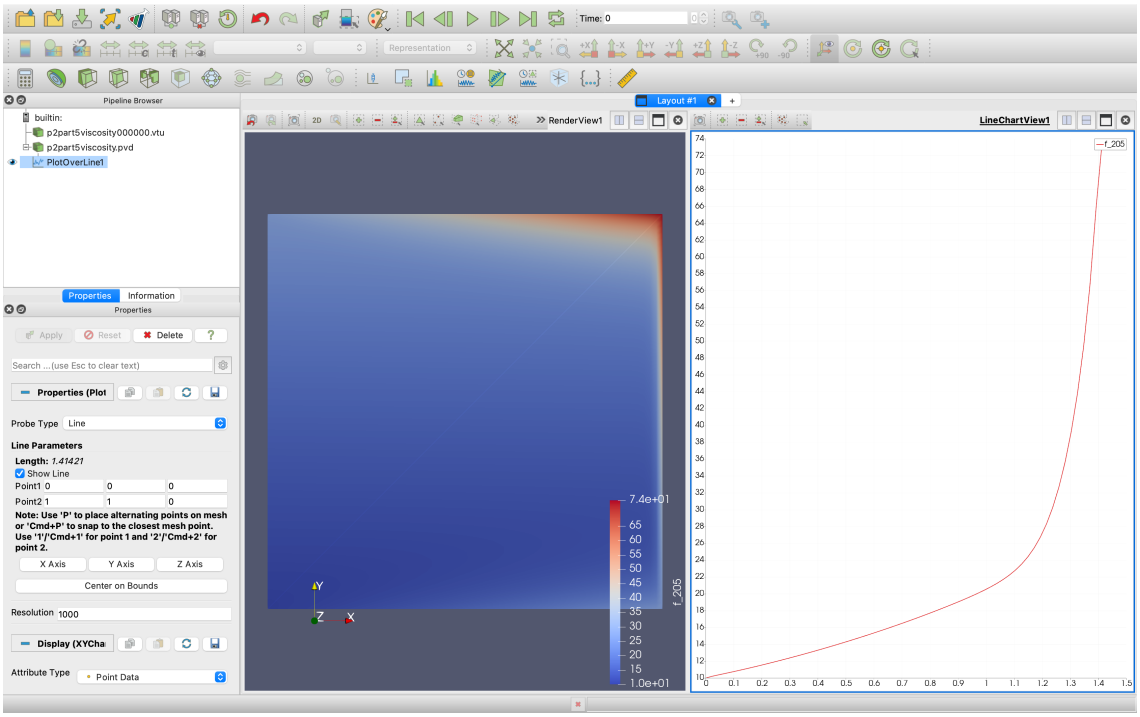
In order to achieve stabilization, we can modify the diffusivity of the problem. Specifically we may change μ to $\mu* = \mu(1 + \mathbb{P}e)$. This way, the equation reads

$$\mu \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \mu \mathbb{P}e \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \int_\Omega (\vec{\beta} \cdot \vec{\nabla} c) v = 0$$

Therefore, we redefine the biliear form as

$$a(c, v) = \mu \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \frac{|\vec{\beta}| h}{2} \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \int_\Omega (\vec{\beta} \cdot \vec{\nabla} c) v$$

The code was modified and executed. The following is the stabilized result.



Note that by changing from μ to $\mu*$, we are solving a problem with a different diffusivity.

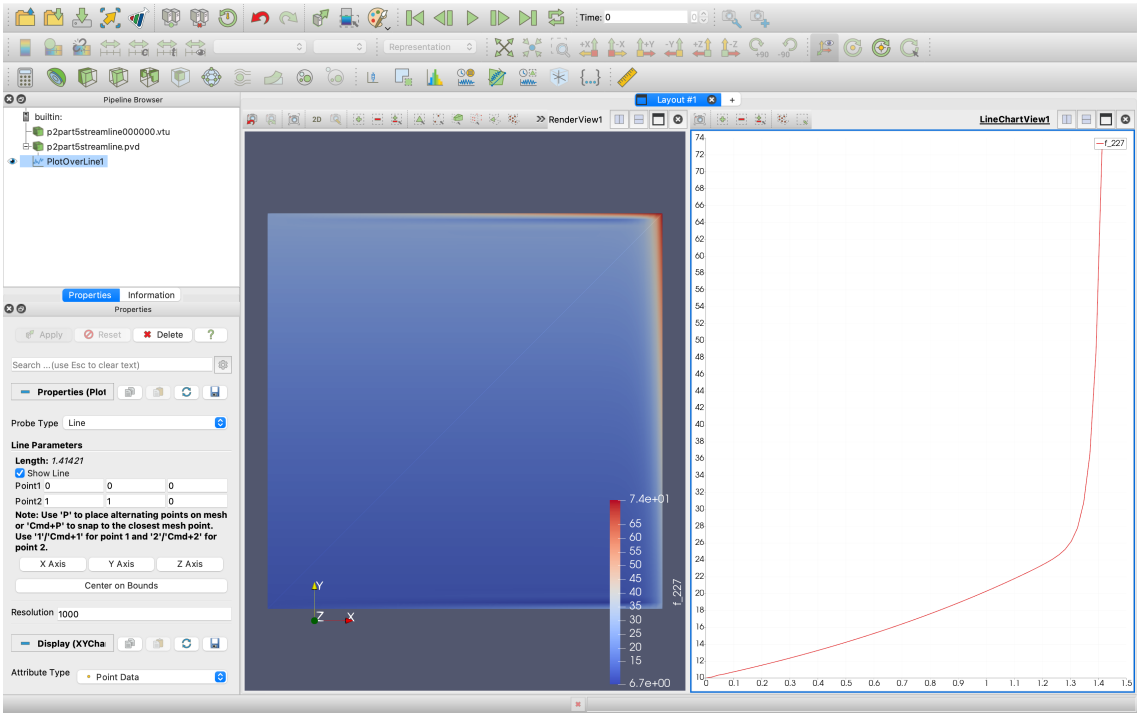
13 Streamline diffusion method

Note that we were using an artificial diffusion term proportional to $\int_\Omega \nabla v \cdot \nabla c$, which adds a useless isotropic stabilization. Therefore, it is preferable to use a term proportional to $\int_\Omega (\nabla v \cdot \vec{\beta})(\nabla u \cdot \vec{\beta})$, as this one contains projections onto $\vec{\beta}$, the advection direction.

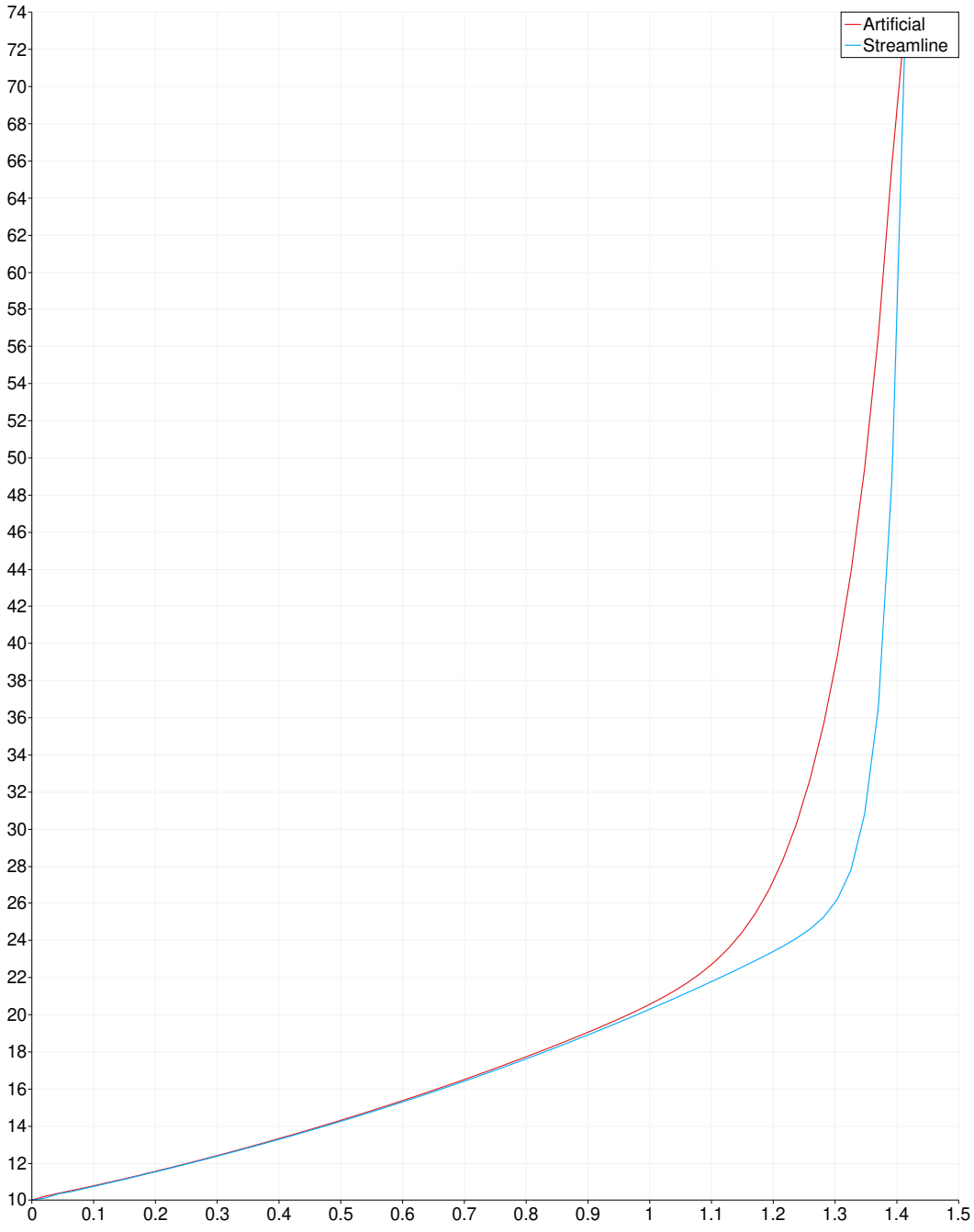
Specifically, allowing $\delta = 1$ be a parameter, we modify the previous bilinear form to

$$a(c, v) = \mu \int_\Omega \vec{\nabla} v \cdot \vec{\nabla} c + \frac{\delta h}{|\vec{\beta}|} \int_\Omega (\vec{\nabla} v \cdot \vec{\beta})(\vec{\nabla} c \cdot \vec{\beta}) + \int_\Omega (\vec{\beta} \cdot \vec{\nabla} c) v$$

The code was modified and executed. The following is the stabilized result.



As mentioned before, the viscosity method adds unnecessary stability in the direction perpendicular to $\vec{\beta}$, while the streamline method only targets this specific direction. Given this, we may expect more accurate results using this last one. For instance, we can notice that the artificial viscosity method overestimates the boundary layer compared to the streamline diffusion method, as can be seen in the following graph.



14 Appendix

Here you can find the codes that we used for this assignment.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sp
import scipy.sparse.linalg
from scipy import integrate

def mapd(xh,xi,xip1):
    x = xi + (xip1-xi)*xh
    j = xip1-xi
    return x,j

def mapi(x,xi,xip1):
    xh = (x-xi)/(xip1-xi)
    ji = 1/(xip1-xi)
    return xh, ji

"""
P1
"""
def phi01(xh):
    # reference function 0 in hat space
    return 1. - xh

def dphi01(xh):
    # reference function 0 in hat space
    return -1. + 0.*xh

def phi11(xh):
    # reference function 1 in hat space
    return xh

def dphi11(xh):
    # reference function 1 in hat space
    return 1.+ 0.*xh

"""
Functions
"""
def f(x):
    return  1.0 + 0.*x

def mu(x):
    return 0.01 + 0.*x

def beta(x):
    return -1. + 0.*x

def sigma(x):
    return 0. + 0.*x

```

```
def bc(a,b):
    ul = 0.0
    ur = 0.0
    return ul, ur

def exact(x, mu=mu(0), alpha=beta(0)):
    u_ex = (1/alpha)*(x - ((np.exp(alpha*x/mu)-1)/(np.exp(alpha/mu)-1)))
    du_ex = (1/alpha)*(1-((alpha*np.exp(alpha*x/mu))/(mu*(np.exp(alpha/mu)
   -1))))
    return u_ex, du_ex

"""
Local Assembly
"""
def loc_asmb1(element,mesh,wg,xhg):
    A_l =[[0.,0.],
          [0.,0.]]
    b_l =[[0.],
          [0.]]
    a = mesh[element]
    b = mesh[element+1]
    x,j = mapd(xhg,a,b)
    ji = 1/j
    A_l[0][0] = sum(wg*(mu(x)*dphi01(xhg)*dphi01(xhg)*ji**2 + \
                        beta(x)*dphi01(xhg)*phi01(xhg)*ji   + \
                        sigma(x)*phi01(xhg)*phi01(xhg))*j)

    A_l[0][1] = sum(wg*(mu(x)*dphi11(xhg)*dphi01(xhg)*ji**2 + \
                        beta(x)*dphi11(xhg)*phi01(xhg)*ji   + \
                        sigma(x)*phi11(xhg)*phi01(xhg))*j)

    A_l[1][0] = sum(wg*(mu(x)*dphi01(xhg)*dphi11(xhg)*ji**2 +\
                        beta(x)*dphi01(xhg)*phi11(xhg)*ji   + \
                        sigma(x)*phi01(xhg)*phi11(xhg))*j)

    A_l[1][1] = sum(wg*(mu(x)*dphi11(xhg)*dphi11(xhg)*ji**2 +\
                        beta(x)*dphi11(xhg)*phi11(xhg)*ji   + \
                        sigma(x)*phi11(xhg)*phi11(xhg))*j)

    b_l[0] =  sum(wg*(f(x)*phi01(xhg))*j)
    b_l[1] =  sum(wg*(f(x)*phi11(xhg))*j)
    return A_l, b_l

def mu_upw(x,h):
    return mu(x)+np.abs(beta(x))*h/2

def mu_sg(x,h):
    Pe = np.abs(beta(x))*h/(2*mu(x))
    return mu(x)*(Pe+2*Pe/(np.exp(2*Pe)-1))

def loc_asmb1UPW(element,mesh,wg,xhg):
    A_l =[[0.,0.],
          [0.,0.]]
    b_l =[[0.],
```

```
            [0.]]
    a = mesh[element]
    b = mesh[element+1]
    h = b-a
    x,j = mapd(xhg,a,b)
    ji = 1/j
    A_l[0][0] = sum(wg*(mu_upw(x,h)*dphi01(xhg)*dphi01(xhg)*ji**2 + \
                        beta(x)*dphi01(xhg)*phi01(xhg)*ji    + \
                        sigma(x)*phi01(xhg)*phi01(xhg))*j)
    A_l[0][1] = sum(wg*(mu_upw(x,h)*dphi11(xhg)*dphi01(xhg)*ji**2 + \
                        beta(x)*dphi11(xhg)*phi01(xhg)*ji    + \
                        sigma(x)*phi11(xhg)*phi01(xhg))*j)
    A_l[1][0] = sum(wg*(mu_upw(x,h)*dphi01(xhg)*dphi11(xhg)*ji**2 +\
                        beta(x)*dphi01(xhg)*phi11(xhg)*ji    + \
                        sigma(x)*phi01(xhg)*phi11(xhg))*j)
    A_l[1][1] = sum(wg*(mu_upw(x,h)*dphi11(xhg)*dphi11(xhg)*ji**2 +\
                        beta(x)*dphi11(xhg)*phi11(xhg)*ji    + \
                        sigma(x)*phi11(xhg)*phi11(xhg))*j)
    b_l[0] =  sum(wg*(f(x)*phi01(xhg))*j)
    b_l[1] =  sum(wg*(f(x)*phi11(xhg))*j)
    return A_l, b_l

"""
Error Computation
"""
def error_h1l2(x,wg,xhg,u):
    Np1 = np.size(x)
    errl2 = 0.0
    errh1 = 0.0
    for el in range(0,Np1-1):
        xg, j = mapd(xhg, x[el], x[el+1])
        ji = 1/j
        ue,due = exact(xg)
        errl2 += sum(wg*j*(u[el]*phi01(xhg)+u[el+1]*phi11(xhg)-ue)**2)
        errh1 += sum(wg*j*(u[el]*dphi01(xhg)*ji+u[el+1]*dphi11(xhg)*ji-due
   )**2)
    errh1 += errl2
    errl2 = np.sqrt(errl2)
    errh1 = np.sqrt(errh1)
    return errl2, errh1

def plot_fine_u(x,q,wg,xhg,u):
    Np1 = np.size(x)
    for el in range(0,Np1-1):
        xx = np.linspace(x[el],x[el+1],q+1)
        xh,ji = mapi(xx,x[el],x[el+1])
        uc = u[el]*phi01(xh) + u[el+1]*phi11(xh)
        uex,duex = exact(xx)
        if el == 0:
            plt.plot(xx,uc, 'b', label='Numerical')
            plt.plot(xx,uex,'r--', label='Exact')
        else:
            plt.plot(xx,uc, 'b')
            plt.plot(xx,uex,'r--')
```

```
157
158     plt.title(f'Solution with alpha={beta(0)}, mu={mu(0)}, h={h}')
159     plt.legend()
160     plt.show()
161     return 0
162
163 """
164 MAIN
165 """
166 # INTEGRATION
167 # Gauss-Legendre (default interval is [-1, 1])
168 deg = 10
169 xr, w = np.polynomial.legendre.leggauss(deg+1)
170 # Translate x values from the interval [-1, 1] to [0, 1]
171 xhg = 0.5*(xr + 1)
172 wg = w/2
173
174 # PROBLEM DEFINITION
175 a = 0
176 b = 1
177 h = 0.01
178 N = int((b-a)/h)
179 uL, uR = bc(a,b)
180 #print(N)
181
182 # MESHING
183 x = np.linspace(0,1,N+1)
184 #print(x)
185 rhs = np.zeros(N+1)
186
187 A =sp.diags([0., 0., 0.], [-1, 0, 1], shape=[N+1, N+1], format = 'csr')
188
189 # ASSEMBLY
190 for el in range(0,N):
191     A_l, b_l = loc_asmb1(el,x,wg,xhg)
192     A[el,el] += A_l[0][0]
193     A[el,el+1] += A_l[0][1]
194     A[el+1,el] += A_l[1][0]
195     A[el+1,el+1] += A_l[1][1]
196     rhs[el] += b_l[0]
197     rhs[el+1] += b_l[1]
198 #    print(A)
199 #    print(rhs)
200
201 # boundary conditions
202 aux = 1.
203 A[0, 1] = 0.
204 A[-1, -2] = 0.
205 A[0, 0] = aux
206 A[-1, -1] = aux
207 rhs[0] = uL*aux
208 rhs[-1] = uR*aux
209
210 # SOLVING
```

```
# Linear system solving
u = sp.linalg.spsolve(A, rhs)

# POSTPROCESSING
# Error computing
q = 100 # quotient fine/coarse

plot_fine_u(x,q,wg,xhg,u)
```

Listing 1: Code used to solve u directly in Problem 1.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sp
import scipy.sparse.linalg
from scipy import integrate

def mapd(xh,xi,xip1):
    x = xi + (xip1-xi)*xh
    j = xip1-xi
    return x,j

def mapi(x,xi,xip1):
    xh = (x-xi)/(xip1-xi)
    ji = 1/(xip1-xi)
    return xh, ji

"""
P1
"""
def phi01(xh):
    # reference function 0 in hat space
    return 1. - xh

def dphi01(xh):
    # reference function 0 in hat space
    return -1. + 0.*xh

def phi11(xh):
    # reference function 1 in hat space
    return xh

def dphi11(xh):
    # reference function 1 in hat space
    return 1.+ 0.*xh

"""
Functions
"""
halo_alpha = 1
halo_mu = 0.1

def f(x, myalpha=halo_alpha, mymu=halo_mu):
    return  np.exp(-myalpha*x/mymu)
```

```
44
45 def mu(x):
46     return halo_mu + 0.*x
47
48 def beta(x):
49     return -halo_alpha + 0.*x
50
51 def sigma(x):
52     return 0. + 0.*x
53
54 def bc(a,b):
55     ul = 0.0
56     ur = 0.0
57     return ul, ur
58
59 def exact(x, mu=mu(0), alpha=-beta(0)):
60     u_ex = (1/alpha)*(((1-np.exp(-alpha*x/mu))/(1-np.exp(alpha/mu))) + (x*
    np.exp(-alpha*x/mu)))
61     du_ex = (1/alpha)*(1-((alpha*np.exp(alpha*x/mu))/(mu*(np.exp(alpha/mu)
    -1))))
62     return u_ex, du_ex
63
64 def exact2(x, mu=mu(0), alpha=-beta(0)):
65     u_ex = (1/alpha)*(x - ((np.exp(alpha*x/mu)-1)/(np.exp(alpha/mu)-1)))
66     du_ex = (1/alpha)*(1-((alpha*np.exp(alpha*x/mu))/(mu*(np.exp(alpha/mu)
    -1))))
67     return u_ex, du_ex
68
69 """
70 Local Assembly
71 """
72 def loc_asmb1(element,mesh,wg,xhg):
73     A_l =[[0.,0.],
74           [0.,0.]]
75     b_l =[[0.],
76           [0.]]
77     a = mesh[element]
78     b = mesh[element+1]
79     x,j = mapd(xhg,a,b)
80     ji = 1/j
81     A_l[0][0] = sum(wg*(mu(x)*dphi01(xhg)*dphi01(xhg)*ji**2 + \
82                         beta(x)*dphi01(xhg)*phi01(xhg)*ji   + \
83                         sigma(x)*phi01(xhg)*phi01(xhg))*j)
84
85     A_l[0][1] = sum(wg*(mu(x)*dphi11(xhg)*dphi01(xhg)*ji**2 + \
86                         beta(x)*dphi11(xhg)*phi01(xhg)*ji   + \
87                         sigma(x)*phi11(xhg)*phi01(xhg))*j)
88
89     A_l[1][0] = sum(wg*(mu(x)*dphi01(xhg)*dphi11(xhg)*ji**2 +\
90                         beta(x)*dphi01(xhg)*phi11(xhg)*ji   + \
91                         sigma(x)*phi01(xhg)*phi11(xhg))*j)
92
93     A_l[1][1] = sum(wg*(mu(x)*dphi11(xhg)*dphi11(xhg)*ji**2 +\
94                         beta(x)*dphi11(xhg)*phi11(xhg)*ji   + \
```

```
                          sigma(x)*phi11(xhg)*phi11(xhg))*j)

    b_l[0] =  sum(wg*(f(x)*phi01(xhg))*j)
    b_l[1] =  sum(wg*(f(x)*phi11(xhg))*j)
    return A_l, b_l


def mu_upw(x,h):
    return mu(x)+np.abs(beta(x))*h/2


def mu_sg(x,h):
    Pe = np.abs(beta(x))*h/(2*mu(x))
    return mu(x)*(Pe+2*Pe/(np.exp(2*Pe)-1))


def loc_asmb1UPW(element,mesh,wg,xhg):
    A_l =[[0.,0.],
          [0.,0.]]
    b_l =[[0.],
          [0.]]
    a = mesh[element]
    b = mesh[element+1]
    h = b-a
    x,j = mapd(xhg,a,b)
    ji = 1/j
    A_l[0][0] = sum(wg*(mu_upw(x,h)*dphi01(xhg)*dphi01(xhg)*ji**2 + \
                        beta(x)*dphi01(xhg)*phi01(xhg)*ji    + \
                        sigma(x)*phi01(xhg)*phi01(xhg))*j)
    A_l[0][1] = sum(wg*(mu_upw(x,h)*dphi11(xhg)*dphi01(xhg)*ji**2 + \
                        beta(x)*dphi11(xhg)*phi01(xhg)*ji    + \
                        sigma(x)*phi11(xhg)*phi01(xhg))*j)
    A_l[1][0] = sum(wg*(mu_upw(x,h)*dphi01(xhg)*dphi11(xhg)*ji**2 +\
                        beta(x)*dphi01(xhg)*phi11(xhg)*ji    + \
                        sigma(x)*phi01(xhg)*phi11(xhg))*j)
    A_l[1][1] = sum(wg*(mu_upw(x,h)*dphi11(xhg)*dphi11(xhg)*ji**2 +\
                        beta(x)*dphi11(xhg)*phi11(xhg)*ji    + \
                        sigma(x)*phi11(xhg)*phi11(xhg))*j)
    b_l[0] =  sum(wg*(f(x)*phi01(xhg))*j)
    b_l[1] =  sum(wg*(f(x)*phi11(xhg))*j)
    return A_l, b_l


"""
Error Computation
"""
def error_h1l2(x,wg,xhg,u):
    Np1 = np.size(x)
    errl2 = 0.0
    errh1 = 0.0
    for el in range(0,Np1-1):
        xg, j = mapd(xhg, x[el], x[el+1])
        ji = 1/j
        ue,due = exact(xg)
```

```
        errl2 += sum(wg*j*(u[el]*phi01(xhg)+u[el+1]*phi11(xhg)-ue)**2)
        errh1 += sum(wg*j*(u[el]*dphi01(xhg)*ji+u[el+1]*dphi11(xhg)*ji-due)**2)
    errh1 += errl2
    errl2 = np.sqrt(errl2)
    errh1 = np.sqrt(errh1)
    return errl2, errh1


def plot_fine_u(x,q,wg,xhg,u):
    Np1 = np.size(x)
    for el in range(0,Np1-1):
        xx = np.linspace(x[el],x[el+1],q+1)
        xh,ji = mapi(xx,x[el],x[el+1])
        uc = u[el]*phi01(xh) + u[el+1]*phi11(xh)
        uex,duex = exact(xx)
        if el == 0:
            plt.plot(xx,uc, 'b', label='Numerical')
            plt.plot(xx,uex,'r--', label='Exact')
        else:
            plt.plot(xx,uc, 'b')
            plt.plot(xx,uex,'r--')

    plt.title(f'Solution for rho with alpha={-beta(0)}, mu={mu(0)}, h={h}')
    plt.legend()
    plt.show()
    return 0

def plot_fine_u2(x,q,wg,xhg,u):
    Np1 = np.size(x)
    for el in range(0,Np1-1):
        xx = np.linspace(x[el],x[el+1],q+1)
        xh,ji = mapi(xx,x[el],x[el+1])
        uc = u[el]*phi01(xh) + u[el+1]*phi11(xh)
        uc = uc*np.exp(-beta(0)*xx/mu(0))
        uex,duex = exact2(xx)
        if el == 0:
            plt.plot(xx,uc, 'b', label='Numerical')
            plt.plot(xx,uex,'r--', label='Exact')
        else:
            plt.plot(xx,uc, 'b')
            plt.plot(xx,uex,'r--')

    plt.title(f'Solution for u with alpha={-beta(0)}, mu={mu(0)}, h={h}')
    plt.legend()
    plt.show()
    return 0

def plot_fine_du(x,q,wg,xhg,u):
    Np1 = np.size(x)
    for el in range(0,Np1-1):
        xx = np.linspace(x[el],x[el+1],q+1)
        xh,ji = mapi(xx,x[el],x[el+1])
```

```
        uc = u[el]*dphi01(xh)*ji + u[el+1]*dphi11(xh)*ji
        uex,duex = exact(xx)
        if el == 0:
            plt.plot(xx,uc, 'b', label='Numerical')
            plt.plot(xx,duex,'r--', label='Exact')
        else:
            plt.plot(xx,uc, 'b')
            plt.plot(xx,duex,'r--')
    plt.title(f'Derivative of solution with alpha={-beta(0)}, mu={mu(0)}, h={h}')
    plt.legend()
    plt.show()
    return 0

"""
MAIN
"""
# INTEGRATION
# Gauss-Legendre (default interval is [-1, 1])
deg = 10
xr, w = np.polynomial.legendre.leggauss(deg+1)
# Translate x values from the interval [-1, 1] to [0, 1]
xhg = 0.5*(xr + 1)
wg = w/2

# PROBLEM DEFINITION
a = 0
b = 1
h = 0.01
N = int((b-a)/h)
uL, uR = bc(a,b)

# MESHING
x = np.linspace(0,1,N+1)
rhs = np.zeros(N+1)
A =sp.diags([0., 0., 0.], [-1, 0, 1], shape=[N+1, N+1], format = 'csr')

# ASSEMBLY
for el in range(0,N):
    A_l, b_l = loc_asmb1(el,x,wg,xhg)
    A[el,el] += A_l[0][0]
    A[el,el+1] += A_l[0][1]
    A[el+1,el] += A_l[1][0]
    A[el+1,el+1] += A_l[1][1]
    rhs[el] += b_l[0]
    rhs[el+1] += b_l[1]

# boundary conditions
aux = 1.
A[0, 1] = 0.
A[-1, -2] = 0.
A[0, 0] = aux
A[-1, -1] = aux
rhs[0] = uL*aux
```

```
rhs[-1] = uR*aux

# SOLVING
# Linear system solving
u = sp.linalg.spsolve(A, rhs)

# POSTPROCESSING
# Error computing
q = 100 # quotient fine/coarse

plot_fine_u(x,q,wg,xhg,u)
plot_fine_u2(x,q,wg,xhg,u)
```

Listing 2: Code used to solve for ρ in Problem 1.

```
from __future__ import print_function
from fenics import *
import matplotlib.pyplot as plt
import numpy as np

# Create mesh and define function space
mesh = UnitSquareMesh(150, 150)
hmax = mesh.hmax()
print(hmax)

V = FunctionSpace(mesh, 'P', 1)

mun = 0.1
mu = Constant(mun)

betan = 10.0
beta = Constant((betan,0.))

print("Peclet",betan*hmax/(2*mun))

# Define boundary condition
u_D = Expression('10.*exp(x[0]+x[1])',degree=4)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(0.0)
delta = 1.0

betamag = betan
h = hmax
sigma =  Constant(0.0)
a = mu*dot(nabla_grad(u), nabla_grad(v))*dx + dot(beta,nabla_grad(u))*v*dx
   + sigma*u*v*dx + delta/betamag*h*(inner(beta,nabla_grad(u))*inner(beta,
```

```
    nabla_grad(v)))*dx
L = f*v*dx
set_log_level(LogLevel.ERROR)

# Compute solution
u = Function(V)
solve(a==L, u, bc)
info(parameters,True)

# Save solution to file in VTK format
vtkfile = File('Lake/p2part5streamline.pvd')
vtkfile << u
```

Listing 3: Code used to solve Problem 2.