MEK4250 Obligatory Assignment 1

Daniel Steeneveldt

Spring 2025

Exercise 5.6 Consider the eigenvalues of the operators, L_1, L_2, and L_3, where $L_1 u = u_x$, $L_2 u = -\alpha u_{xx}$, $\alpha = 1.0e^{-5}$, and $L_3 = L_1 + L_2$, with homogeneous Dirichlet conditions. For which of the operators are the eigenvalues positive and real? Repeat the exercise with $L_1 = xu_x$.

Solution 5.6 L_1**:** The eigenvalue problem is $u_x = \lambda u$. The soluion is $u(x) = Ce^{\lambda x}$. With the boundary conditions $u(0) = u(1) = 0$, we get $C = 0$ and $\lambda = 0$. Thus, the eigenvalues of L_1 are $\lambda = 0$.

L_2**:** The eigenvalue problem is $L_2 u = \lambda u$. We have looked at this in lectures. It's easy to see that sine satesfies the eigenfunction equation.

$$L_2 \sin(n\pi x) = \alpha n^2 \pi^2 \sin(n\pi x) = \lambda \sin(n\pi x).$$

Thus, the positive real eigenvalues of L_2 are $\lambda = n^2\pi^2\alpha$. (e^{Cx} has negative eigenvalues, and e^{iCx} has complex eigenvalues.)

L_3**:** The eigenvalue problem is $u_x - \alpha u_{xx} = \lambda u$ or $u_x - \alpha u_{xx} - \lambda u = 0$. We look for a solution on the form $u(x) = e^{rx}$.

$$\begin{aligned} re^{rx} - \alpha r^2 e^{rx} - \lambda e^{rx} &= 0 \\ r - \alpha r^2 - \lambda &= 0 \\ \alpha r^2 - r + \lambda &= 0 \\ r &= \frac{1 \pm \sqrt{1 - 4\alpha\lambda}}{2\alpha} \\ r_1 = \frac{1 + \sqrt{1 - 4\alpha\lambda}}{2\alpha}, \quad r_2 &= \frac{1 - \sqrt{1 - 4\alpha\lambda}}{2\alpha} \end{aligned}$$

The general solution is then

$$u(x) = C_1 e^{r_1 x} + C_2 e^{r_2 x}.$$

Inserting the boundary conditions $u(0) = u(1) = 0$ gives

$$\begin{aligned} u(0) = C_1 + C_2 = 0 \quad &\Longleftrightarrow \quad C_2 = -C_1, \\ u(1) = C_1 e^{r_1} + C_2 e^{r_2} = 0 \quad &\Longleftrightarrow \quad C_1(e^{r_1} - e^{r_2}) = 0. \end{aligned}$$

Looking at the non trivial case $C_1 \neq 0$, we get

$$e^{r_1} - e^{r_2} = 0 \quad \Longleftrightarrow \quad e^{r_1} = e^{r_2} \quad \Longleftrightarrow \quad e^{r_1 - r_2} = 1.$$

This gives us the condition $r_1 - r_2 = i2\pi k$. Since $e^{i2\pi k} = \cos(2\pi k) + i\sin(2\pi k) = 1$ for all $k \in \{0, 1, ...\}$.

$k = 0$**:**

$$\begin{aligned} r_1 - r_2 = \frac{1 + \sqrt{1 - 4\alpha\lambda}}{2\alpha} - \frac{1 - \sqrt{1 - 4\alpha\lambda}}{2\alpha} = \frac{2\sqrt{1 - 4\alpha\lambda}}{2\alpha} = 0. \\ \sqrt{1 - 4\alpha\lambda} = 0 \quad \Longleftrightarrow \quad 1 - 4\alpha\lambda = 0 \quad \Longleftrightarrow \quad \lambda = \frac{1}{4\alpha} \end{aligned}$$

$k > 0$**:**

$$\begin{aligned} r_1 - r_2 = \frac{2\sqrt{1 - 4\alpha\lambda}}{2\alpha} &= i2\pi k \\ \sqrt{1 - 4\alpha\lambda} &= i\pi k\alpha \end{aligned}$$

Since it's imaginary we have that $1 - 4\alpha\lambda < 0$ and we can write $\sqrt{1 - 4\alpha\lambda} = i\sqrt{4\alpha\lambda - 1}$. Since $\sqrt{-x} = i\sqrt{x}$ for positve x.

$$\begin{aligned} i\sqrt{4\alpha\lambda - 1} = i\pi k\alpha \quad \Longleftrightarrow \quad \sqrt{4\alpha\lambda - 1} = \pi k\alpha \quad \Longleftrightarrow \quad 4\alpha\lambda - 1 = \pi^2 k^2 \alpha^2 \\ \lambda = \frac{1 + \pi^2 k^2 \alpha^2}{4\alpha} \end{aligned}$$

Those are then the eigenvalues of L_3.

Modified $L_1 u = x u_x$**:** The eigenvalue problem is $x u_x = \lambda u$.

$$\begin{aligned} x u_x = \lambda u \quad &\Longleftrightarrow \quad \frac{u_x}{u} = \frac{\lambda}{x} \\ ln(u) = \lambda ln(x) + C \quad &\Longleftrightarrow \quad u = C x^{\lambda} \end{aligned}$$

With the boundary conditions $u(0) = u(1) = 0$, we get $C = 0$ so there are no eigenvalues for $L_1 = x u_x$.

Modified $L_3 = xu_x - \alpha u_{xx}$: The eigenvalue problem is $xu_x - \alpha u_{xx} = \lambda u$. Solving this analyticaly seems difficult, so we will solve it numerically, using fem.

Weak form

$$\int_0^1 (xu_x - \alpha u_{xx})v\,dx = \lambda \int_0^1 uv\,dx, \quad u, v \in H_0^1$$

Integrate the double derivative by parts

$$\begin{aligned}\int_0^1 -\alpha u_{xx} v\,dx &= \int_0^1 \alpha u_x v_x\,dx - [\alpha uv]_0^1 \\ &= \int_0^1 \alpha u_x v_x\,dx\end{aligned}$$

The weak form is then

$$\int_0^1 (xu_x v + \alpha u_x v_x)\,dx = \lambda \int_0^1 uv\,dx$$

We let $u = \sum u_j N_j$ where N_j are the basis trial functions. Here we use the same test functions. Giving us a systen of equations

$$\sum u_j \int_0^1 (xN_i' N_j + \alpha N_i' N_j')\,dx = \lambda \sum \int_0^1 N_i N_j\,dx$$

Which we can write on matrix form $Au = \lambda M u$ where A and M are the stiffness and mass matrecies.

```python
from dolfinx import mesh, fem
from dolfinx.fem import petsc
import ufl
from mpi4py import MPI
import numpy as np
from scipy.linalg import eig
from scipy.sparse import csr_matrix
import numpy as np
import matplotlib.pyplot as plt

N = 300
left = 0.0
right = 1.0
```

```python
domain = mesh.create_interval(MPI.COMM_WORLD, N, [left, right])
V = fem.functionspace(domain, ("Lagrange", 1))

u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)

alpha = 1.0e-5

x = ufl.SpatialCoordinate(domain)[0]

a = (x * u.dx(0) * v + alpha * ufl.inner(ufl.grad(u), ufl.grad(v)))
↪  * ufl.dx
m = u * v * ufl.dx

def boundary(x):
    return np.logical_or(np.isclose(x[0], left), np.isclose(x[0],
    right))

boundary_dofs = fem.locate_dofs_geometrical(V, boundary)
bc = fem.dirichletbc(0.0, boundary_dofs, V)


A = petsc.assemble_matrix(fem.form(a), bcs=[bc])
A.assemble()
M = petsc.assemble_matrix(fem.form(m), bcs=[bc])
M.assemble()

Ai, Aj, Av = A.getValuesCSR()
Mi, Mj, Mv = M.getValuesCSR()
A_dense = csr_matrix((Av, Aj, Ai)).toarray()
M_dense = csr_matrix((Mv, Mj, Mi)).toarray()
evals, _ = eig(A_dense, M_dense)
plt.plot(evals.real, evals.imag, "o")
plt.show()

finite_evals = evals[np.isfinite(evals)]
real_positive = [float((np.real(ev))) for ev in finite_evals if
np.isreal(ev) and ev.real > 0]
print("Number of positive real eigenvalues:", len(real_positive))
print("Positive real eigenvalues:", real_positive)
```

The above code with N=300 have the following output

```
Number of positive real eigenvalues: 5
Positive real eigenvalues:[2.1999999999963507, 1.000049939411666, 5.7999999
```

For the mesh size of 1000 we got 25 eigenvalues. Only got 1 for mesh size 100, I guess the point is that we get eigenvalues whenever we have diffusion, however I do not understand how eigenvalues correspond to stability.

Exercise 6.1 Show that the conditions (6.15)-(6.17) are satisfied for $V_h = H_0^1(\Omega)$ and $Q_h = L^2(\Omega)$.

Solution 6.1 Let's begin by restating the conditions (6.15)-(6.17). Boundedness of a:

$$a(u_h, v_h) \le C_1 \|u_h\|_{V_h} \|v_h\|_{V_h}, \quad \forall u_h, v_h \in V_h. \tag{6.15}$$

Boundedness of b:

$$b(u_h, q_h) \le C_2 \|u_h\|_{V_h} \|q_h\|_{Q_h}, \quad \forall u_h \in V_h, q_h \in Q_h. \tag{6.16}$$

Coercivity of a:

$$a(u_h, u_h) \ge C_3 \|u_h\|_{V_h}^2, \quad \forall u_h \in Z_h. \tag{6.17}$$

$Z_h = \{u_h \in V_h : b(u_h, q_h) = 0, \forall q_h \in Q_h\}$.

Recall that poincare tells us

$$\|u\|_{L^2} \le C \|\nabla u\|_{L^2}$$

Which gives equivalence of norms in H^1 and the seminorm

$$\|\nabla u\|_{L^2} \le \|u\|_{H^1} \le \sqrt{C^2+1} \|\nabla u\|_{L^2}$$

Boundedness of a:

$$\begin{aligned} a(u_h, v_h) &= \int_\Omega \nabla u_h : \nabla v_h \, dx \\ &\le \|\nabla u_h\|_{L^2} \|\nabla v_h\|_{L^2} \\ &\le \|u_h\|_{H^1} \|v_h\|_{H^1}. \end{aligned}$$

On the second line we have used the Cauchy Schwarz (CS) inequality, which holds for $\langle f, g \rangle = \int_\Omega f : g \, dx$. Also $\|\nabla u\|_{L^2}^2 = \int_\Omega \nabla u : \nabla u \, dx$.

Boundedness of b:

$$\begin{aligned} b(p_h, v_h) &= \int_\Omega p_h \nabla \cdot v_h \, dx \\ &\leq \|p_h\|_{L^2} \|\nabla \cdot v_h\|_{L^2}, \quad \text{CS integral of scalar functions} \end{aligned}$$

$$\begin{aligned} \|\nabla \cdot v_h\|_{L^2}^2 &= \int \Big(\sum_{j=1}^d \partial_{x_j} u_{h,j} \Big)^2 dx \\ &\leq \int \sum_{j=1}^d d \Big(\partial_{x_j} u_{h,j} \Big)^2_{L^2} dx, \quad \text{CS on integrand with 1} \\ &= d \sum_{j=1}^d \|\partial_{x_j} u_{h,j}\|_{L^2}^2 \leq d \sum_{i=1}^d \sum_{j=1}^d \|\partial_{x_i} u_{h,j}\|_{L^2}^2 = d \|\nabla u_h\|_L^2. \end{aligned}$$

$$\begin{aligned} b(p_h, v_h) &\leq \|p_h\|_{L^2} \|\nabla \cdot v_h\|_{L^2} \\ &\leq \|p_h\|_{L^2} \sqrt{d} \|\nabla v_h\|_{L^2} \\ &\leq \sqrt{d} \|p_h\|_{L^2} \|v_h\|_{H^1}. \end{aligned}$$

Coercivity of a:

We get the coercivity of a by the Poincare inequality.

$$\begin{aligned} a(u_h, u_h) &= \int_\Omega \nabla u_h : \nabla u_h \, dx \\ &= \|\nabla u_h\|_{L^2}^2 \\ &\geq \frac{1}{(1+C)^2} \|u_h\|_{H^1}^2. \end{aligned}$$

Exercise 6.2 Show that the conditions (6.15)-(6.17) are satisfied for Taylor- Hood and Mini discretizations. (Note that Crouzeix-Raviart is non-conforming so it is more difficult to prove these conditions for this case.)

Solution 6.2 Taylor-Hood: In the book they are desribed as such

$$u : N_i = a_i + b_i x + c_i y + d_i xy + e_i x^2 + f_i y^2,$$
$$p : L_i = k_i + l_i x + m_i y.$$

The book also notes that these are generalized to higher orders by having $V_h \subset \mathcal{P}_k$ and $Q_h \subset \mathcal{P}_{k-1}$. I'll stick to this.

Since the trial functions are polynomials they are in H^1 and L^2. Thus by the previous exercise the conditions are satisfied.

Mini:

The book describes the velocity and pressure to be linear, exept for an added bubble with an added degree of freedom for the velocity.

$$u : N_i = a_i + b_i x + c_i y + d_i xy(1 - x - y),$$

$$p : L_i = k_i + l_i x + m_i y.$$

$N_i \in H^1$ and $L_i \in L^2$, so the conditions are satisfied.

Exercise 6.6 In the previous problem, the solution was a second-order polynomial in the velocity and first order in the pressure. We may therefore obtain the exact solution, making it difficult to check the order of convergence for higher-order methods with this solution. In this exercise, you should therefore implement the problem:

$$\begin{aligned} u &= (\sin(\pi y), \cos(\pi x)), \\ p &= \sin(2\pi x), \\ f &= -\Delta u - \nabla p. \end{aligned}$$

Test whether the approximation is of the expected order for the following element pairs: $P_4 - P_3$ $P_4 - P_2$ $P_3 - P_2$ $P_3 - P_1$

Solution 6.6 Weak form

$$\int_\Omega -\nabla \cdot (\nabla u + pI) \cdot v \, dx = \int_\Omega f \cdot v \, dx, \quad \forall v \in V,$$

Dot product is linear, lets look at the part involving u, $-(\nabla \cdot \nabla u) \cdot v$. We have the following identiy, similar to the classical product rule in elementary calculus $\nabla \cdot (\nabla u v) = (\nabla \cdot \nabla u) \cdot v + \nabla u : \nabla v$

$$\begin{aligned} \int_\Omega -\nabla \cdot \nabla u \cdot v \, dx + \int_\Omega \nabla \cdot (\nabla u v) \, dx &= \int_\Omega \nabla u : \nabla v dx \\ \int_\Omega -\nabla \cdot \nabla u \cdot v \, dx + \int_{\partial\Omega} \nabla u n \cdot v \, ds &= \int_\Omega \nabla u : \nabla v dx \\ \int_\Omega -\nabla \cdot \nabla u \cdot v \, dx &= \int_\Omega \nabla u : \nabla v dx - \int_{\partial\Omega} \nabla u n \cdot v \, ds \end{aligned}$$

The part involving p would be $-\nabla \cdot pI \cdot v = -\nabla p \cdot v$ We use the same identity again $\nabla \cdot (pv) = (\nabla p) \cdot v + p\nabla \cdot v$

$$\int_\Omega -\nabla p \cdot v\, dx = \int_\Omega p\nabla \cdot v\, dx - \int_{\partial\Omega} pv\, ds, \quad \forall v \in V_0$$

Giving us the weak form (we also need the divergence free condition)

$$\begin{aligned}
\int_\Omega \nabla u : \nabla v dx - \int_{\partial\Omega} \nabla u \cdot nv\, ds + \int_\Omega p\nabla \cdot v\, dx - \int_{\partial\Omega} pv\, ds &= \int_\Omega fv\, dx \\
\int_\Omega \nabla u : \nabla v\, dx - \int_{\partial\Omega} (\nabla u + Ip) \cdot n \cdot v\, ds + \int_\Omega p\nabla \cdot v\, dx &= \int_\Omega fv\, dx \\
\int_\Omega \nabla u : \nabla v\, dx + \int_\Omega p\nabla \cdot v\, dx &= \int_\Omega fv\, dx + \int_{\partial\Omega} (\nabla u + Ip) \cdot n \cdot v\, ds \\
\int_\Omega \nabla u : \nabla v\, dx + \int_\Omega p\nabla \cdot v\, dx &= \int_\Omega fv\, dx + \int_{\partial\Omega} h \cdot v\, ds \\
\int_\Omega q\nabla \cdot u\, dx &= 0, \quad \forall v \in V_0,
\end{aligned}$$

We also have the boundary conditions

$$u = (\sin(\pi y), \cos(\pi x)) \quad \text{on} \quad x \in \partial\Omega_D$$

$$\begin{aligned}
h = (\nabla u + Ip) \cdot n &= \begin{pmatrix} 0 & \pi\cos(\pi y) \\ -\pi\sin(\pi x) & 0 \end{pmatrix} + \begin{pmatrix} \sin(2\pi x) & 0 \\ 0 & \sin(2\pi x) \end{pmatrix} \cdot n \\
&= \begin{pmatrix} \sin(2\pi x) & \pi\cos(\pi y) \\ -\pi\sin(\pi x) & \sin(2\pi x) \end{pmatrix} \cdot n
\end{aligned}$$

If we have that the right wall is the Neumann boundary, we get that $n = (1, 0)$ and it's easy to verify that $h = (0, 0)$. Thus we do not need to add any boundary source term to the weak form. If, however, we have that the Nemuann boundary is the bottom wall we need to incldude the source h from the analytical solution.

I decided to only write one script for both exercises 6.6 and 6.7, since they are very similar.

Exercise 6.7 Implement the Stokes problem with the analytical solution $u = (\sin(\pi y), \cos(\pi x))$, $p = \sin(2\pi x)$, and $f = -\Delta u - \nabla p$, on the unit square.

Consider the case where Dirichlet boundary conditions are imposed on the sides $x = 0$, $x = 1$, and $y = 1$, while a Neumann condition is used on the remaining side (this avoids the singular system associated with either pure Dirichlet or pure Neumann problems). Then, determine the order of approximation of the wall shear stress on the side $x = 0$. The wall shear stress is given by $\nabla u \cdot t$, where $t = (0, 1)$ is the tangent vector along $x = 0$.

Solution 6.7 The part not explained would be the error calculation. From the book we have the following error estimate.

$$\|u - u_h\|_1 + \|p - p_h\|_0 \leq C\, h^k \|u\|_{k+1} + D\, h^{\ell+1} \|p\|_{\ell+1}$$

Where k is the order of the velocity approximation and ℓ is the order of the pressure approximation. Even tough the H1 and H1 seminorms are equivalent, I did not like that not all constants where on the right hand side. I decided to calculate the error in the H1 norm for the velocity and the L2 norm for the pressure.

All solutions had $\ell \leq k$ and satesfied the above error estimate. Giving

$$\begin{aligned} \|u - u_h\|_1 + \|p - p_h\|_0 &\leq C\, h^k \|u\|_{k+1} + D\, h^{\ell+1} \|p\|_{\ell+1} \\ &\leq h^{\ell+1} \left(C\, h^{k-\ell-1} \|u\|_{k+1} + D\, \|p\|_{\ell+1} \right) \\ &\leq h^{\ell+1} C^* \end{aligned}$$

To get the convergence rate we calculate the left hand side for different mesh sizes h. Each error bounds are then on the form $E_i = C^* h_i^r$, where r is the convergence rate. We solve for r by $E_{i-1} = C^* h_{i-1}^r$ giving us

$$r = \frac{\log(E_{i-1}) - \log(E_i)}{\log(h_{i-1}) - \log(h_i)}$$

Everything seemed to converge to the expected $\ell + 1$ convergence rate. Except fot the wall shear stress which converged by k, the polynomial degree of the velocity approximation. This is because we are calculate the wall shear stress on the dirichlet boundary, where we have interpolated the exact error. So we are efficiently calculating the error in polynomal interpolation.

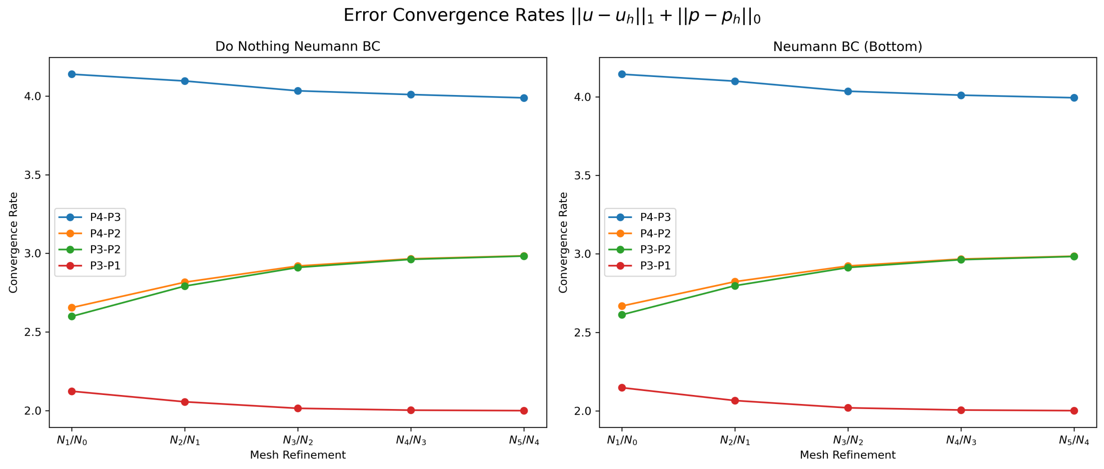
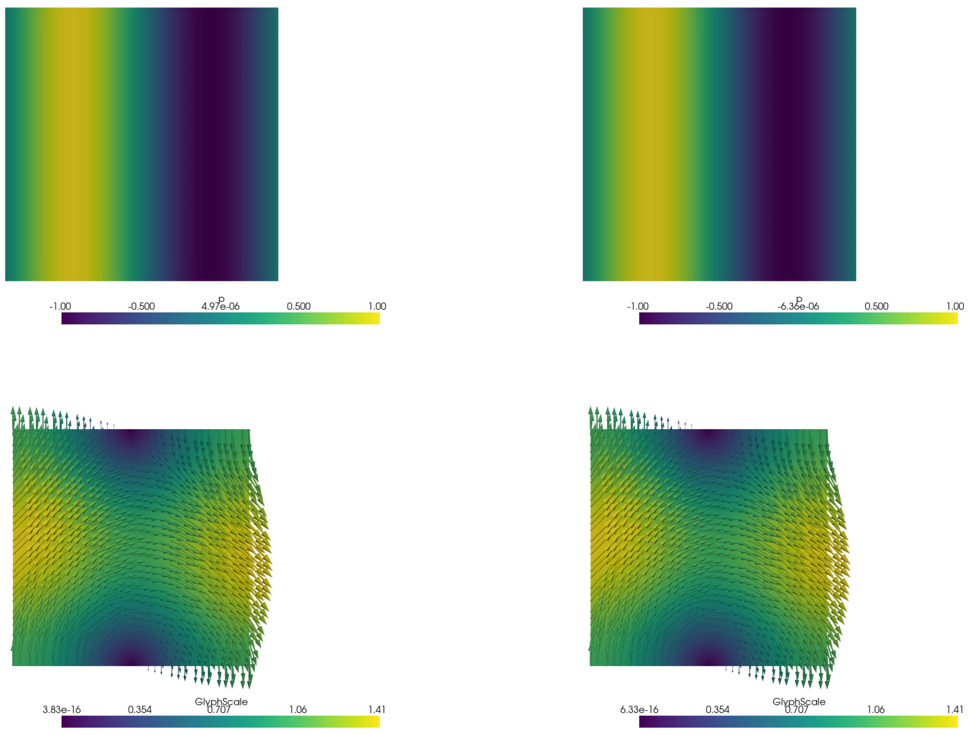


Figure 1: Convergence for different element pairs. $N = [4, 8, 16, 32, 64, 128]$. Left is exercise 6.6 and right is Neumann on bottom Exercise 6.7.



(a) Exercise 6.6: Pressure and Velocity (b) Exercise 6.7: Pressure and Velocity

Figure 2: Pressure and Velocity Figures for Exercises 6.6 and 6.7.

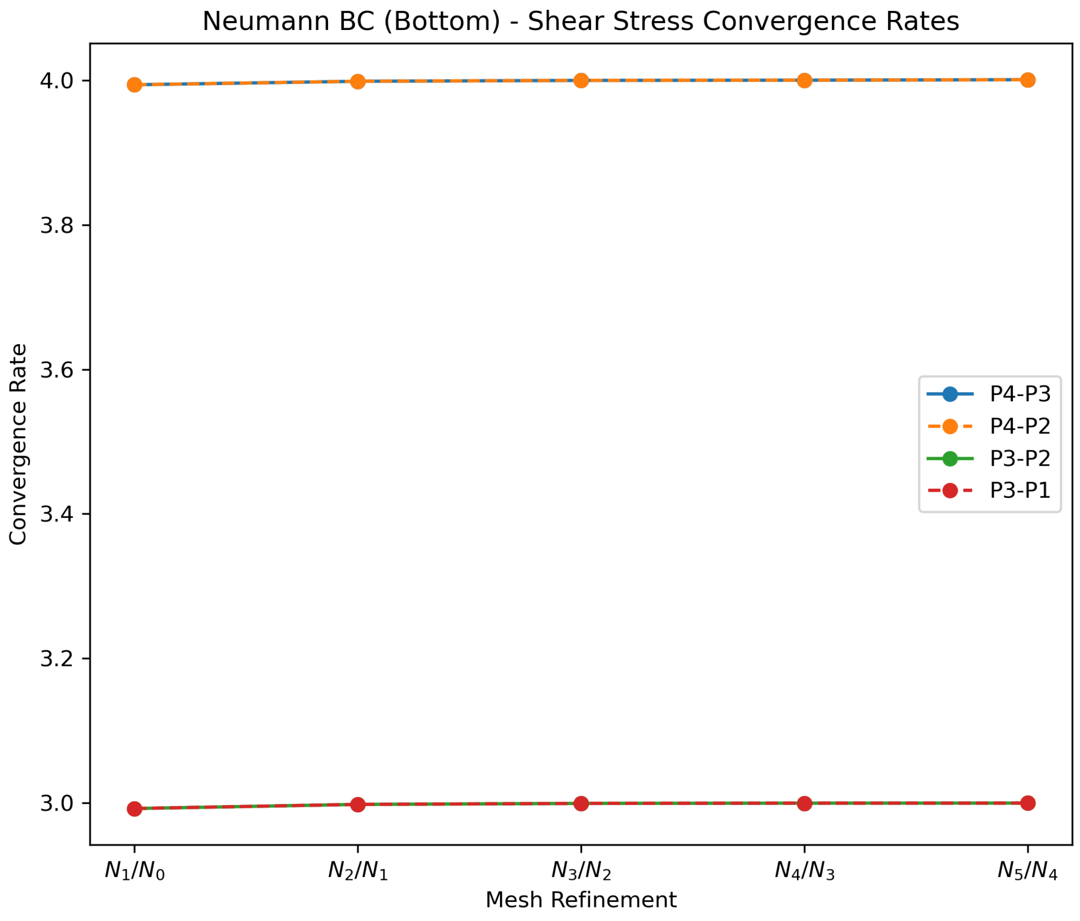


Figure 3: Wall shear stress for different element pairs.

Below is the code for exercise 6.6 and 6.7. I've put the plotting code at the very end.

```python
from collections.abc import Callable
from dolfinx import fem, mesh
import basix.ufl
import ufl
from mpi4py import MPI
import numpy as np
from plotter import plot_shear_stress_rates, plot_convergence_rates
from dolfinx.fem.petsc import LinearProblem



def boundary_functions_factory(neumann_boundary: str = "bottom") -> tuple[Callable[[np.ndarray], np.ndarray],
                                                                                          ↪ Callabl
                                                                                          ↪ np.ndar
    """
    Factory function to create functions for identifying Dirichlet and Neumann boundaries
    based on the boundary name.

    Returns vectorized Neumann and Dirichlet boundary tag functions.

    """
    all_boundary_conditions = {'left' : lambda x : np.isclose(x[0], 0.0),
                               'bottom' : lambda x : np.isclose(x[1], 0.0),
                               'right' : lambda x: np.isclose(x[0], 1.0),
                               'top' : lambda x : np.isclose(x[1], 1.0)
    }
    on_neumann = all_boundary_conditions.pop(neumann_boundary)
    def on_dirichlet(x):
        return np.logical_or.reduce([func(x) for func in all_boundary_conditions.values()])

    return on_dirichlet, on_neumann

def u_exact_numpy(x: np.ndarray) -> np.ndarray:
    """
    Exact solution for the velocity field.
    Used for interpolating onto the function space.

    """
    return np.sin(np.pi * x[1]), np.cos(np.pi * x[0])

def p_exact_numpy(x: np.ndarray) -> np.ndarray:
    """
    Exact solution for the pressure field.
    Used for interpolating onto the function space.

    """
    return np.sin(2*np.pi * x[0])

def u_exact_ufl(x: ufl.SpatialCoordinate) -> ufl.Coefficient:
    """
    Exact solution for the velocity field.
    Used for symbolicaly defining the exact solution, in the
    ↪  residual form of the problem.

    """
    return ufl.as_vector([ufl.sin(ufl.pi * x[1]), ufl.cos(ufl.pi *
    ↪  x[0])])


def p_exact_ufl(x: ufl.SpatialCoordinate) -> ufl.Coefficient:
    """
    Exact solution for the pressure field.
    Used for symbolicaly defining the exact solution, in the
    ↪  residual form of the problem.

    """
    return ufl.sin(2 * ufl.pi * x[0])


```

```
def setup_system(W: fem.FunctionSpace, enforce_neumann=True):
    """
    Setup the variational problem for the Stokes equations. The
    ↪  math of the weak form is defined here.

    """
    domain = W.mesh

    u, p = ufl.TrialFunctions(W) #linear combs of basis functions
    v, q = ufl.TestFunctions(W)

    x = ufl.SpatialCoordinate(domain)


    f = -ufl.div(ufl.grad(u_exact_ufl(x))) -
    ↪  ufl.grad(p_exact_ufl(x))
    F = ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
    F += ufl.inner(p, ufl.div(v)) * ufl.dx
    F += ufl.inner(ufl.div(u), q) * ufl.dx
    F -= ufl.inner(f, v) * ufl.dx

    if enforce_neumann:
        n = ufl.FacetNormal(domain)
        h = (ufl.grad(u_exact_ufl(x)) +
        ↪  p_exact_ufl(x)*ufl.Identity(len(u_exact_ufl(x)))) * n
        F -= ufl.inner(h, v) * ufl.ds
        #we can integrate over all of ds since v=0 on the
        ↪  dirichlet boundary, which is the remainding part of
        ↪  the boundary
    else:
        pass #Do nothing boundary condtion

    a, L = ufl.system(F)

    return a, L


def solve_stokes(N, polypair, neumann_boundary ="right",
↪  enforce_neumann=False, plot=False, savefig=False, savename=""):
    """
    Solve the Stokes problem on a unit square using mixed finite
    ↪  elements.

    Parameters:
    N : int
        Number of cells in each direction.
    polypair : tuple
        Polynomial degree pair for the velocity and pressure
        ↪  spaces.
    neumann_boundary : str
        Name of the Neumann boundary.
    enforce_neumann : bool
        Whether to enforce the Neumann boundary condition.
    .
    .
    Returns:
    uh : dolfinx.fem.Function
        Approximate velocity field.
    ph : dolfinx.fem.Function
        Approximate pressure field.
    u_exact : dolfinx.fem.Function
        Exact velocity field.
    p_exact : dolfinx.fem.Function
        Exact pressure field.

    """

    on_dirichlet, on_neumann =
    ↪  boundary_functions_factory(neumann_boundary)

    #-----setup domain and function space----
    p_u, p_p = polypair
    domain = mesh.create_unit_square(MPI.COMM_WORLD, N, N)
    el_u = basix.ufl.element("Lagrange", domain.basix_cell(), p_u,
    ↪  shape=(domain.geometry.dim,))
    el_p = basix.ufl.element("Lagrange", domain.basix_cell(), p_p)
    el_mixed = basix.ufl.mixed_element([el_u, el_p])
    W = fem.functionspace(domain, el_mixed)

    #-----setup variational problem----
    a, L = setup_system(W, enforce_neumann)


    #------Dirichlet boundary conditions----
    W0 = W.sub(0)
    V, V_to_W0 = W0.collapse()

    u_exact = fem.Function(V)
    u_exact.interpolate(u_exact_numpy)

    dir_facets = mesh.locate_entities_boundary(domain,
    ↪  domain.topology.dim - 1, on_dirichlet)
    combined_dofs = fem.locate_dofs_topological((W0, V),
    ↪  domain.topology.dim - 1, dir_facets)
    bc = fem.dirichletbc(u_exact, combined_dofs, W0)
    bcs = [bc]

    #-----solve the problem----
    problem = LinearProblem(
        a,
        L,
        bcs=bcs,
        petsc_options={
            "ksp_type": "preonly",
            "pc_type": "lu",
            "pc_factor_mat_solver_type": "mumps",
        },
    )
    wh = problem.solve()

    return wh




def raised_difference(exact: Callable[[np.ndarray], np.ndarray],
                      approx: fem.Function,
                      degree_raise: int = 3) -> fem.Function:
    """ Get exact solution for the same quadrature points as
    ↪  approximation, interpolate both to a higher
```

```
            space and return the difference between the exact and
            ↪  approximate solutions.

    Based on
    ↪  https://jsdokken.com/dolfinx-tutorial/chapter4/convergence.html
    ↪  with help from August Femtehjell

    Args:
        exact (Callable[[np.ndarray], np.ndarray]): The exact
        ↪  solution.
        approx (dolfinx.fem.Function): The approximate solution.
        degree_raise (int, optional): The degree raise for the
        ↪  space. Defaults to 3.

    Returns:
        dolfinx.fem.Function: The error function.
    """
    # Get the function space and mesh
    V = approx.function_space
    domain = V.mesh
    degree = V.ufl_element().degree
    family = V.ufl_element().family_name
    shape = V.value_shape

    # Create a higher-order function space
    Ve = fem.functionspace(domain, (family, degree + degree_raise,
    ↪  shape))

    # Interpolate the exact solution to the higher-order function
    ↪  space
    u_ex = fem.Function(Ve)
    u_ex.interpolate(exact)

    # Interpolate the approximate solution to the higher-order
    ↪  function space
    u_h = fem.Function(Ve)
    u_h.interpolate(approx)

    difference = fem.Function(Ve)
    # Compute the difference between the exact and approximate
    ↪  solutions
```

```python
    difference.x.array[:] = u_ex.x.array - u_h.x.array

    return difference


def L2_error(exact: Callable[[np.ndarray], np.ndarray], approx: fem.Function, comm=None, measure=ufl.dx):
    """
    Calculate the L2 error between the exact and approximate solutions.

    """
    if comm is None:
        comm = approx.function_space.mesh.comm
    diff = raised_difference(exact, approx)
    error_form = fem.form(ufl.inner(diff, diff) * measure)
    error_value = np.sqrt(comm.allreduce(fem.assemble_scalar(error_form), op=MPI.SUM))
    return error_value


def H1_seminorm_error(exact: Callable[[np.ndarray], np.ndarray], approx: fem.Function, measure=ufl.dx):
    """
    Calculate the H1 error (gradient error) between the exact and approximate solutions.
    returns:
    error_value : float
        ||grad(exact - approx)||_L2
    """
    comm = approx.function_space.mesh.comm
    diff = raised_difference(exact, approx)
    error_form = fem.form(ufl.inner(ufl.grad(diff), ufl.grad(diff)) * measure)
    error_value = np.sqrt(comm.allreduce(fem.assemble_scalar(error_form), op=MPI.SUM))
    return error_value

```

```python

def calculate_shear_stress(u_exact: Callable[[np.ndarray],
↪ np.ndarray], uh: fem.Function, neumann_boundary="left"):
    """
    Calculate the L2 error in the shear stress on a specified
    ↪ Neumann boundary.

    returns:
    shear_error : float
        ||grad(u_exact - uh) * t||_L2

    """

    _, on_neumann = boundary_functions_factory(neumann_boundary)

    domain = uh.function_space.mesh
    neumann_facets = mesh.locate_entities_boundary(
        domain,
        domain.topology.dim - 1,
        on_neumann
    )

    mt = mesh.meshtags(
        domain,
        domain.topology.dim - 1,
        neumann_facets,
        np.full_like(neumann_facets, 0, dtype=np.int32)
    )

    ds = ufl.Measure("ds", domain=uh.function_space.mesh,
    ↪ subdomain_data=mt)
    # Define normal and tangent vectors
    diff = raised_difference(u_exact, uh)

    n = ufl.FacetNormal(diff.function_space.mesh)
    t = ufl.as_vector([n[1], -n[0]])


    error_form = fem.form(ufl.inner(ufl.grad(diff)*t,
    ↪ ufl.grad(diff)*t) * ds(0))
    comm = uh.function_space.mesh.comm
    error_value = np.sqrt(comm.allreduce(fem.assemble_scalar(error_form), op=MPI.SUM))
    return error_value



def experiment_poly_pairs(Ns: list[int], polypairs: list[tuple[int, int]],
                            enforce_neumann: bool = False, neumann_boundary: str = "right"):
    """
    Run experiments for a set of polynomial pairs.

    If enforce_neumann is False, it runs the standard (do nothing) experiment,
    computing a single error (pressure + velocity error).
    If enforce_neumann is True, it runs the experiment with a Neumann boundary condition
    (using the specified neumann_boundary) and computes two errors:
      - error_solution: combined solution error (including an extra L2 velocity error)
      - error_shear: error in the computed shear stress.

    Returns:
      For do nothing: (Es, Hs, rates)
      For Neumann: (Es_solution, Es_shear, Hs, rates_solution, rates_shear)
    """
    num_N = len(Ns)
    num_poly = len(polypairs)
    Es = np.zeros((num_N, num_poly))
    Hs = np.zeros((num_N, num_poly))

    if enforce_neumann:
        Es_shear = np.zeros((num_N, num_poly))

    for j, poly in enumerate(polypairs):
        for i, N in enumerate(Ns):
            if enforce_neumann:
                wh = solve_stokes(N, poly, enforce_neumann=True, neumann_boundary=neumann_boundary)
                uh = wh.sub(0).collapse()
                ph = wh.sub(1).collapse()
                error_solution = (L2_error(p_exact_numpy, ph) +
                                  H1_seminorm_error(u_exact_numpy, uh) +
                                  L2_error(u_exact_numpy, uh))
                error_shear = calculate_shear_stress(u_exact_numpy, uh, neumann_boundary="left")
                Es[i, j] = error_solution
                Es_shear[i, j] = error_shear
            else:
                wh = solve_stokes(N, poly)
                uh = wh.sub(0).collapse()
                ph = wh.sub(1).collapse()
                error = L2_error(p_exact_numpy, ph) + H1_seminorm_error(u_exact_numpy, uh) + L2_error(u_exact_numpy, uh)
                Es[i, j] = error
            Hs[i, j] = 1.0 / N


    if enforce_neumann:
        rates_solution = np.log(Es[:-1, :] / Es[1:, :]) / np.log(Hs[:-1, :] / Hs[1:, :])
        rates_shear = np.log(Es_shear[:-1, :] / Es_shear[1:, :]) / np.log(Hs[:-1, :] / Hs[1:, :])
        return Es, Es_shear, Hs, rates_solution, rates_shear
    else:
        rates = np.log(Es[:-1, :] / Es[1:, :]) / np.log(Hs[:-1, :] / Hs[1:, :])
        return Es, Hs, rates

if __name__ == "__main__":
    Ns = [4, 8, 16, 32, 64, 128]
    polypairs = [(4, 3), (4, 2), (3, 2), (3, 1)]

    # Ex 6.6
    Es_dn, Hs_dn, rates_dn = experiment_poly_pairs(Ns, polypairs, enforce_neumann=False)

    # Ex 6.7
    Es_neu, Es_shear_neu, Hs_neu, rates_sol_neu, rates_shear_neu = experiment_poly_pairs(
        Ns, polypairs, enforce_neumann=True, neumann_boundary="bottom"
    )

    if MPI.COMM_WORLD.rank == 0:

        print("Mean convergence rates for Do Nothing BC:")
        print(np.mean(rates_dn, axis=0))
        print("Mean convergence rates for Neumann BC (Solution):")
        print(np.mean(rates_sol_neu, axis=0))
        print("Mean convergence rates for Neumann BC (Shear Stress):")
        print(np.mean(rates_shear_neu, axis=0))


        plot_convergence_rates(Ns, rates_dn, rates_sol_neu, polypairs)
        plot_shear_stress_rates(Ns, rates_shear_neu, polypairs)

```

Plotting

```python
import pyvista
import dolfinx
import numpy as np
from pathlib import Path
import matplotlib.pyplot as plt

def visualize_mixed(mixed_function: dolfinx.fem.Function, scale=0.1, savefig=False, savename=""):
    """
    Plot a mixed function with a vector and scalar component. Mostly
    compied from dokken tutorial.

    """
    u_c = mixed_function.sub(0).collapse()
    p_c = mixed_function.sub(1).collapse()

    u_grid =
    ↪  pyvista.UnstructuredGrid(*dolfinx.plot.vtk_mesh(u_c.function_space))

    # Pad u to be 3D
    gdim = u_c.function_space.mesh.geometry.dim
    assert len(u_c) == gdim
    u_values = np.zeros((len(u_c.x.array) // gdim, 3),
    ↪  dtype=np.float64)
    u_values[:, :gdim] = u_c.x.array.real.reshape((-1, gdim))

    # Create a point cloud of glyphs
    u_grid["u"] = u_values
    glyphs = u_grid.glyph(orient="u", factor=scale)
    pyvista.set_jupyter_backend("static")
    plotter = pyvista.Plotter()
    plotter.add_key_event("Escape", lambda: plotter.close())
    plotter.add_mesh(u_grid, show_edges=False,
    ↪  show_scalar_bar=False)
    plotter.add_mesh(glyphs)
    plotter.view_xy()
    plotter.show()
    if savefig:
        #check if figs folder exists and create it if not
        folder_path = Path("figs")
        folder_path.mkdir(parents=True, exist_ok=True)
        plotter.screenshot(r"figs/velocity_" + savename + ".png",
        ↪  transparent_background=True)

    p_grid =
    ↪  pyvista.UnstructuredGrid(*dolfinx.plot.vtk_mesh(p_c.function_space))
    p_grid.point_data["p"] = p_c.x.array
    plotter_p = pyvista.Plotter()
    plotter_p.add_mesh(p_grid, show_edges=False)
    plotter_p.view_xy()
    plotter_p.show()
    if savefig:
        plotter_p.screenshot(r"figs/pressure_" + savename + ".png",
         ↪  transparent_background=True)





def plot_convergence_rates(Ns, rates_dn, rates_sol_neu, polypairs):
    """
    Plot convergence rates (solution error) for the two different
    ↪  boundary conditions
    in a single figure with two subplots.

    Left subplot: Do Nothing BC.
    Right subplot: Neumann BC (bottom) solution error.
    """
    fig, axes = plt.subplots(1, 2, figsize=(14, 6))
    fig.suptitle(r"Error Convergence Rates $||u - u_h||_1 + || p -
    ↪  p_h||_0$ ", fontsize=16)
    x_ticks = [f"$N_{{{i+1}}}/N_{{{i}}}$" for i in range(len(Ns) -
    ↪  1)]
    x = range(len(x_ticks))

    # Plot for Do Nothing BC.
    for j, poly in enumerate(polypairs):
        axes[0].plot(x, rates_dn[:, j], marker='o',
         ↪  label=f"P{poly[0]}-P{poly[1]}")
    axes[0].set_title("Do Nothing Neumann BC")
    axes[0].set_xlabel("Mesh Refinement")
    axes[0].set_ylabel("Convergence Rate")
    axes[0].set_xticks(x)
    axes[0].set_xticklabels(x_ticks)
    axes[0].legend()

    # Plot for Neumann BC (Bottom) - solution error.
    for j, poly in enumerate(polypairs):
        axes[1].plot(x, rates_sol_neu[:, j], marker='o',
         ↪  label=f"P{poly[0]}-P{poly[1]}")
    axes[1].set_title("Neumann BC (Bottom)")
```

```python
    axes[1].set_xlabel("Mesh Refinement")
    axes[1].set_ylabel("Convergence Rate")
    axes[1].set_xticks(x)
    axes[1].set_xticklabels(x_ticks)
    axes[1].legend()

    plt.tight_layout()
    plt.savefig("figs/convergence_rates_subplot.png", dpi=300)
    plt.show()
    plt.close()


def plot_shear_stress_rates(Ns, rates_shear_neu, polypairs):
    """
    Plot shear stress convergence rates for the Neumann BC (Bottom)
    for all polynomial pairs.
    """
    fig, ax = plt.subplots(figsize=(7, 6))
    x_ticks = [f"$N_{{{i+1}}}/N_{{{i}}}$" for i in range(len(Ns) - 1)]
    x = range(len(x_ticks))
    linestyle_func = lambda j: '-' if j % 2 == 0 else '--'

    for j, poly in enumerate(polypairs):
        ax.plot(x, rates_shear_neu[:, j], marker='o', label=f"P{poly[0]}-P{poly[1]}", linestyle=linestyle_func(j))
    ax.set_title("Neumann BC (Bottom) - Shear Stress Convergence Rates")
    ax.set_xlabel("Mesh Refinement")
    ax.set_ylabel("Convergence Rate")
    ax.set_xticks(x)
    ax.set_xticklabels(x_ticks)
    ax.legend()

    plt.tight_layout()
    plt.savefig("figs/shear_stress_convergence_rates.png", dpi=300)
    plt.show()
    plt.close()
```