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 $L_1 = x u_x$.

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 form in lectures. It's easy to see that sinus fufills 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 $u(x) = e^{rx}$.

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

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 $x u_x = \lambda u$.

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

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 $Au = \lambda Mu$ where A and M are the stiffness and mass matrix.

```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:

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

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

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

Coercivity of a:

We get the coercivity of a by the Poincare inequality.

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

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

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

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

$$\int_\Omega -\nabla \cdot \nabla u \cdot v \, dx + \int_\Omega \nabla \cdot (\nabla u v) \, dx = \int_\Omega \nabla u : \nabla v dx$$
$$\int_\Omega -\nabla \cdot \nabla u \cdot v \, dx + \int_{\partial\Omega} \nabla u \cdot n v \, ds = \int_\Omega \nabla u : \nabla v dx, \quad \forall v \in V_0$$

We use the same identity again for the pressure term $\nabla \cdot (pv) = (\nabla p) \cdot v + p\nabla \cdot v$

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

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

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

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 parts 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

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

I've put the plotting code at the bottom in a seperate .py file as a i find it less interesting.

```python
from dolfinx import fem, mesh, la
import basix.ufl
import ufl
from mpi4py import MPI
import numpy as np
import scipy.sparse
from plotter import visualize_mixed, loglog_plot, convergence_rate_plot



def boundary_functions_factory(neumann_boundary: str = "bottom"):
    """
    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):
    """
    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):
    """
    Exact solution for the pressure field.
    Used for interpolating onto the function space.

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


```

```python
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)

    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)

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

    x = ufl.SpatialCoordinate(domain)
    u_exact_ufl = ufl.as_vector([ufl.sin(ufl.pi * x[1]), ufl.cos(ufl.pi * x[0])])
    p_exact_ufl = ufl.sin(2 * ufl.pi * x[0])


    f = -ufl.div(ufl.grad(u_exact_ufl)) - ufl.grad(p_exact_ufl)
    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:
        #Neumann boundary condition
        facets = mesh.locate_entities_boundary(domain, domain.topology.dim - 1, on_neumann)
        mt = mesh.meshtags(domain, domain.topology.dim - 1, facets, 0) # passing 0, but i dont want to use tags here integrating over all of ds
        ds = ufl.Measure("ds", domain=domain, subdomain_data=mt)
        n = ufl.FacetNormal(domain)
        F -= ufl.inner((ufl.grad(u_exact_ufl) - p_exact_ufl*ufl.Identity(len(u_exact_ufl))) * n, v) * ds
    else:
        pass #Do nothing boundary condtion

    a, L = ufl.system(F)


    domain.topology.create_connectivity(domain.topology.dim - 1, domain.topology.dim)
    dir_facets = mesh.locate_entities_boundary(domain, domain.topology.dim - 1, on_dirichlet)
```

```python

    W0 = W.sub(0)
    V, V_to_W0 = W0.collapse()

    W1 = W.sub(1)
    Q, Q_to_W1 = W1.collapse()


    u_exact = fem.Function(V)
    p_exact = fem.Function(Q)
    p_exact.interpolate(p_exact_numpy)
    u_exact.interpolate(u_exact_numpy)

    #Dirichlet boundary conditions
    combined_dofs = fem.locate_dofs_topological((W0, V),
    ↪   domain.topology.dim - 1, dir_facets)
    bc = fem.dirichletbc(u_exact, combined_dofs, W0)
    bcs = [bc]

    #Form, create, assemble and solve system
    a_compiled = fem.form(a) #create c code for the form
    L_compiled = fem.form(L)
    A = fem.create_matrix(a_compiled) #create matrix code for the
    ↪   form
    b = fem.create_vector(L_compiled)
    A_scipy = A.to_scipy()
    fem.assemble_matrix(A, a_compiled, bcs=bcs) #actually fill the
    ↪   matrix
    fem.assemble_vector(b.array, L_compiled)
    fem.apply_lifting(b.array, [a_compiled], [bcs])
    b.scatter_reverse(la.InsertMode.add) #tell ghosted vector to
    ↪   add values to local dofs
    bc.set(b.array) #set the boundary condition values to the
    ↪   vector

    A_inv = scipy.sparse.linalg.splu(A_scipy) #lu factorization

    wh = fem.Function(W)
    wh.x.array[:] = A_inv.solve(b.array) #solve Ax=b and put the
    ↪   solution in wh
```

```python
    if plot:
        visualize_mixed(wh, scale=0.1, savefig=savefig, savename=savename)

    uh = wh.sub(0).collapse()
    ph = wh.sub(1).collapse()

    return uh, ph, u_exact, p_exact


def calculate_L2_error(exact, approx, comm=None, measure=ufl.dx):
    """
    Calculate the L2 error between the exact and approximate solutions.

    """
    if comm is None:
        comm = exact.function_space.mesh.comm
    # Define the error form
    error_form = fem.form(ufl.inner(exact - approx, exact - approx) * measure)
    # Assemble the error and perform a global reduction
    error_value = np.sqrt(comm.allreduce(fem.assemble_scalar(error_form), op=MPI.SUM))
    return error_value


def calculate_H1_seminorm_error(exact, approx, measure=ufl.dx):
    """
    Calculate the H1 error (gradient error) between the exact and approximate solutions.
    returns:
    error_value : float
        ||grad(exact - approx)||_L2
    """
    comm = exact.function_space.mesh.comm
    error_form = fem.form(ufl.inner(ufl.grad(exact - approx), ufl.grad(exact - approx)) * measure)
```

```python
    error_value = np.sqrt(comm.allreduce(fem.assemble_scalar(error_form), op=MPI.SUM))
    return error_value


def calculate_shear_stress(u_exact, uh, neumann_boundary="left"):
    """
    Calculate the L2 error in the shear stress on a specified Neumann boundary.

    returns:
    shear_error : float
        ||grad(u_exact) * t - grad(uh) * t||_L2 #could also have done H1 seminorm of u*t,uh*t

    """
    comm = uh.function_space.mesh.comm
    _, on_neumann = boundary_functions_factory(neumann_boundary)
    # Locate facets on the Neumann boundary
    neumann_facets = mesh.locate_entities_boundary(
        uh.function_space.mesh,
        uh.function_space.mesh.topology.dim - 1,
        on_neumann
    )
    # Create meshtags for the Neumann boundary; here, all facets are tagged with 0
    mt = mesh.meshtags(
        uh.function_space.mesh,
        uh.function_space.mesh.topology.dim - 1,
        neumann_facets,
        np.full(len(neumann_facets), 0, dtype=np.int32)
    )
    ds = ufl.Measure("ds", domain=uh.function_space.mesh, subdomain_data=mt)
    # Define normal and tangent vectors
    n = ufl.FacetNormal(uh.function_space.mesh)
    t = ufl.as_vector([n[1], -n[0]])

    shear_exact = ufl.dot(ufl.grad(u_exact), t)
```

```
    shear_approx = ufl.dot(ufl.grad(uh), t)

    return calculate_L2_error(shear_exact, shear_approx, comm=comm, measure=ds)


def experiment_ex66(Ns):
    """
    Run experiment ex66 to evaluate the convergence of the solution for various polynomial pairs.

    """
    polypairs = [(4, 3), (4, 2), (3, 2), (3, 1)]
    num_N = len(Ns)
    num_polypairs = len(polypairs)
    Es = np.zeros((num_N, num_polypairs))
    Hs = np.zeros((num_N, num_polypairs))

    for j, polypair in enumerate(polypairs):
        for i, N in enumerate(Ns):
            uh, ph, u_exact, p_exact = solve_stokes(N, polypair)
            # Compute errors (order: exact, approx)
            error_pressure = calculate_L2_error(p_exact, ph)
            error_velocity = calculate_H1_seminorm_error(u_exact, uh)
            Es[i, j] = error_pressure + error_velocity
            Hs[i, j] = 1.0 / N

    # Compute convergence rates between successive mesh refinements
    rates = np.log(Es[:-1] / Es[1:]) / np.log(Hs[:-1] / Hs[1:])
    mean_rates = np.mean(rates, axis=0)
    print(f"Mean error convergence rates of solution: {mean_rates}")
    return Es, Hs, rates


def experiment_ex67(Ns):
    """
    Run experiment ex67 to evaluate the convergence of both the solution and the shear stress on a Neumann boundary.
```

```python

    """
    neumann_boundary = "left"
    polypair = (3, 2)
    num_N = len(Ns)
    Es = np.zeros((num_N, 2))  # Column 0: solution error; Column 1: shear stress error
    Hs = np.zeros(num_N)

    for i, N in enumerate(Ns):
        uh, ph, u_exact, p_exact = solve_stokes(
            N, polypair, enforce_neumann=True, neumann_boundary=neumann_boundary
        )
        error_solution = calculate_L2_error(p_exact, ph) + calculate_H1_seminorm_error(u_exact, uh) + calculate_L2_error(u_exact, uh)
        error_shear = calculate_shear_stress(u_exact, uh, neumann_boundary=neumann_boundary)
        Es[i, 0] = error_solution
        Es[i, 1] = error_shear
        Hs[i] = 1.0 / N

    rates_sol = np.log(Es[:-1, 0] / Es[1:, 0]) / np.log(Hs[:-1] / Hs[1:])
    rates_shear = np.log(Es[:-1, 1] / Es[1:, 1]) / np.log(Hs[:-1] / Hs[1:])
    return Es, Hs, rates_sol, rates_shear


def test(plot=False, enforce_neumann=False, neumann_boundary='right', savefig=False, savename=""):
    """
    Test the Stokes solver and error calculations, and optionally plot or save the results.

    """
    uh, ph, u_exact, p_exact = solve_stokes(
        10, (3, 2), plot=plot, enforce_neumann=enforce_neumann,
        neumann_boundary=neumann_boundary, savefig=savefig, savename=savename
```

```
    )
    # Compute errors with the convention: exact, approx
    error = calculate_H1_seminorm_error(u_exact, uh) + calculate_L2_error(p_exact, ph)
    comm = uh.function_space.mesh.comm
    shear_error = calculate_shear_stress(u_exact, uh, neumann_boundary='left')
    if comm.rank == 0:
        print(f"Error: {error:.2e}")
        print(f"Shear stress error: {shear_error:.2e}")

if __name__ == "__main__":
    Ns = [2, 4, 8, 16, 32, 64]
    #test(plot=True, savefig=False, savename='66', neumann_boundary="right") #plot
    #test(plot=True, enforce_neumann=True, neumann_boundary='right', savefig=False, savename='67') #plot
    #test(plot=True, enforce_neumann=True, neumann_boundary='bottom', savefig=False, savename='67') #plot

    #quit()
    Es66, Hs66, rates66 = experiment_ex66(Ns)
    Es67, Hs67, rates_sol67, rates_shear67 = experiment_ex67(Ns)

    if MPI.COMM_WORLD.rank == 0:
        print(f"Mean error convergence rates of solutions ex66: {np.mean(rates66, axis=1)}")
        print(f"Mean error convergence rates of solution ex67: {np.mean(rates_sol67)}")
        print(f"Mean error convergence rates of shear stress ex67: {np.mean(rates_shear67)}")

        loglog_plot(Hs67, Es67, Hs66, Es66)
        convergence_rate_plot(Ns, rates66, rates_sol67, rates_shear67)
```

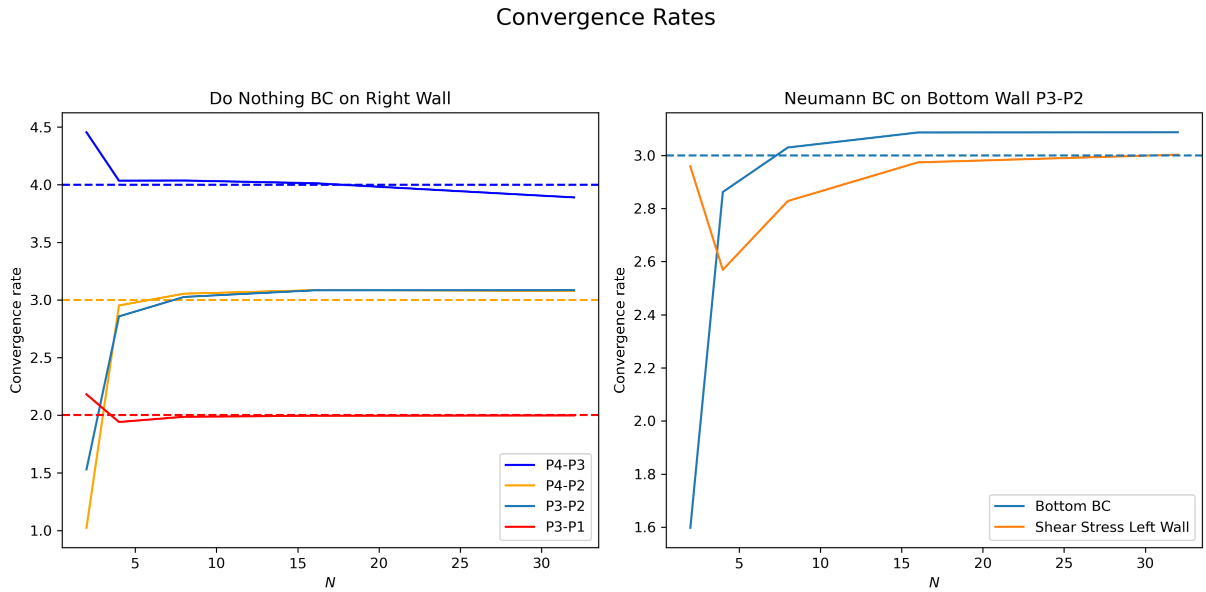
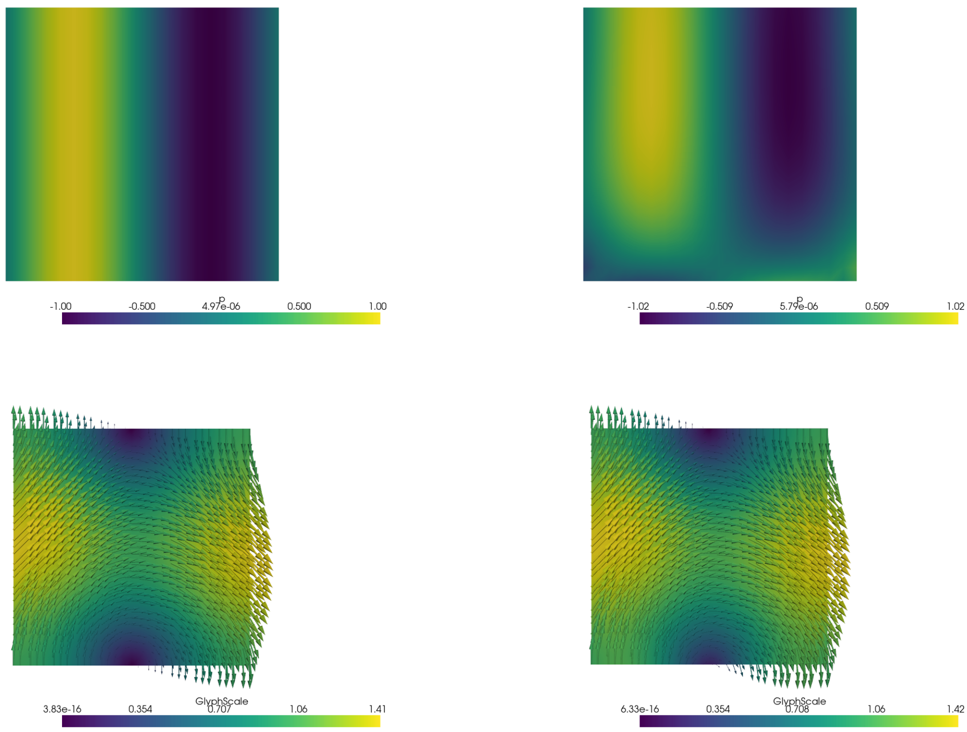


Figure 1: Convergence for different element pairs. Dotted lines show expected convergence rates.



(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.

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=1.0, 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:
```

```python
        #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 loglog_plot(Hs67, Es67, Hs66, Es66):
    """
    Plot log-log error plots for the two different boundary conditions.
    """
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))
    fig.suptitle("Log-Log Error plots", fontsize=16)

    axes[0].set_title("Neumann BC on Bottom Wall P3-P2")
    axes[0].loglog(Hs67, Es67[:, 0], label="Error solution")
    axes[0].loglog(Hs67, Es67[:, 1], label="Error shear stress")
    axes[0].set_xlabel("h")
    axes[0].set_ylabel("Error")
    axes[0].legend()

    Hs_same = Hs66[:, 0]
    axes[1].set_title("Neumann BC (do nothing) on Right Wall")
    axes[1].loglog(Hs_same, Es66[:, 0], label="Error P4-P3")
    axes[1].loglog(Hs_same, Es66[:, 1], label="Error P4-P2")
    axes[1].loglog(Hs_same, Es66[:, 2], label="Error P3-P2")
    axes[1].loglog(Hs_same, Es66[:, 3], label="Error P3-P1")
    axes[1].set_xlabel("h")
```

```python
    axes[1].set_ylabel("Error")
    axes[1].legend()

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


def convergence_rate_plot(Ns, rates66, rates_sol67, rates_shear67):
    """
    Plot convergence rates for the two different boundary conditions.
    """
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))
    fig.suptitle("Convergence Rates", fontsize=16)

    x = np.array(Ns[:-1])

    axes[0].plot(Ns[:-1], rates66[:, 0], label="P4-P3", color="blue")
    axes[0].axhline(y=4, color="blue", linestyle="--")

    axes[0].plot(Ns[:-1], rates66[:, 1], label="P4-P2", color="orange")
    axes[0].axhline(y=3, linestyle="--", color="orange")

    axes[0].plot(Ns[:-1], rates66[:, 2], label="P3-P2")
    axes[0].plot(Ns[:-1], rates66[:, 3], label="P3-P1", color="red")
    axes[0].axhline(y=2, linestyle="--", color="red")
    axes[0].set_ylabel("Convergence rate")
    axes[0].set_xlabel(r"$N$")
    axes[0].legend()
    axes[0].set_title("Do Nothing BC on Right Wall")

    axes[1].plot(Ns[:-1], rates_sol67, label="Bottom BC")
    axes[1].plot(Ns[:-1], rates_shear67, label="Shear Stress Left Wall")
    axes[1].axhline(y=3, linestyle="--")
```

```python
    axes[1].set_ylabel("Convergence rate")
    axes[1].set_xlabel(r"$N$")
    axes[1].legend()
    axes[1].set_title("Neumann BC on Bottom Wall P3-P2")

    plt.tight_layout(rect=[0, 0, 1, 0.93])
    plt.savefig("figs/convergence_rates.png", dpi=300)
    plt.show()
    plt.close()
```