COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

THE FU FOUNDATION SCHOOL OF ENGINEERING AND APPLIED SCIENCES

DEPARTMENT OF MECHANICAL ENGINEERING

MECE: E6106 Finite Element Method for Fluid Flow and Fluid-Structure Interactions

Nicolino Primavera (UNI: ncp2136)

Dr. Vijay Vedula

December 20, 2024

Table of Contents

1. Objective

The objective was to compare the numerically calculated pressure gradient against the analytical value assuming Stokes flow. FEniCS, which is an open-source computing platform for solving partial differential equations (PDEs) using the finite element method (FEM), was used to help simulate the problem using stabilized finite element (FE) method and *inf-sup* conforming finite elements. The problem contains a 2D channel with a height of 1cm and a length of 8cm. The boundary conditions of the problem include a parabolic inflow velocity, a traction free outflow, and no-slip/no penetration walls. The inflow is a Dirichlet condition with a velocity profile, $\frac{u}{U_\infty} = \frac{y}{H}(1-\frac{y}{H})$, where $U_\infty = 1cm/s$ is the velocity at the centerline. The outflow is a traction-free boundary condition, $t_{\hat{n}} = \boldsymbol{T} \cdot \hat{n} = 0$ where $\boldsymbol{T}$ is the Couchy stress tensor given by: $\boldsymbol{T} = -p\mathbf{I} + 2\mu\mathbf{D}$ and $\mathbf{D} = \frac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^{\mathrm{T}})$ is the rate-of-strain tensor (symmetric gradient of velocity). The top/bottom walls are fixed no-slip/no penetration where the fluid velocity ($\mathbf{u}$) = 0.

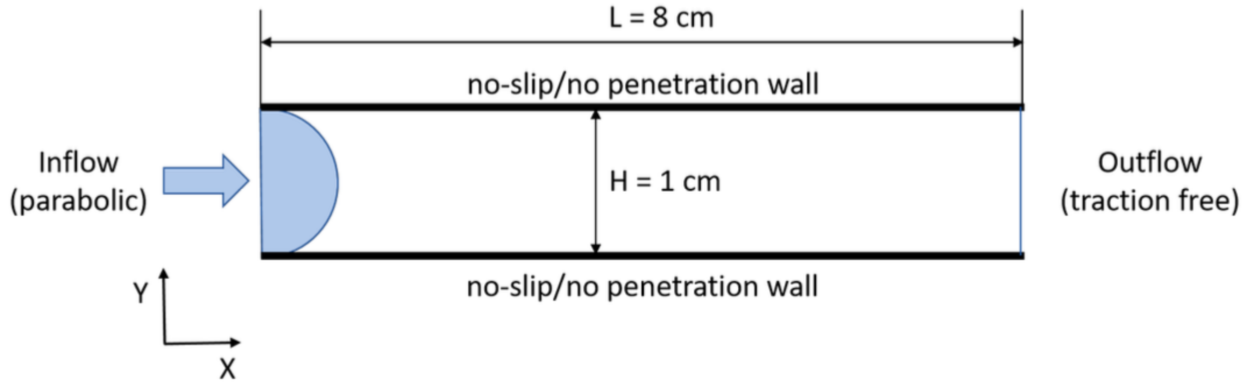


Figure 1: Schematic of the domain and boundary conditions

The physical parameters include:

- incompressible flow, no body force ($\boldsymbol{f^b} = 0$)
- fluid velocity: $\mathbf{u}$
- fluid pressure: p
- fluid density: $\rho = 1g/cm^3$
- fluid viscosity: $\mu = 1g/cm/s$
- Reynolds number: $Re = \frac{\rho U_\infty H}{\mu} = 1$
    - Therefore, stokes flow is a reasonable assumption for the above configuration.

The governing equations for stokes flow:

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} - \boldsymbol{f^b}\right) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}$$

$$\nabla \cdot \boldsymbol{u} = 0 \text{ (incompressibility)}$$

where $\boldsymbol{\sigma_{ij}} = -p\delta_{ij} + \mu\left(u_{i,j} + u_{j,i}\right)$ is the Cauchy stress for the fluid.

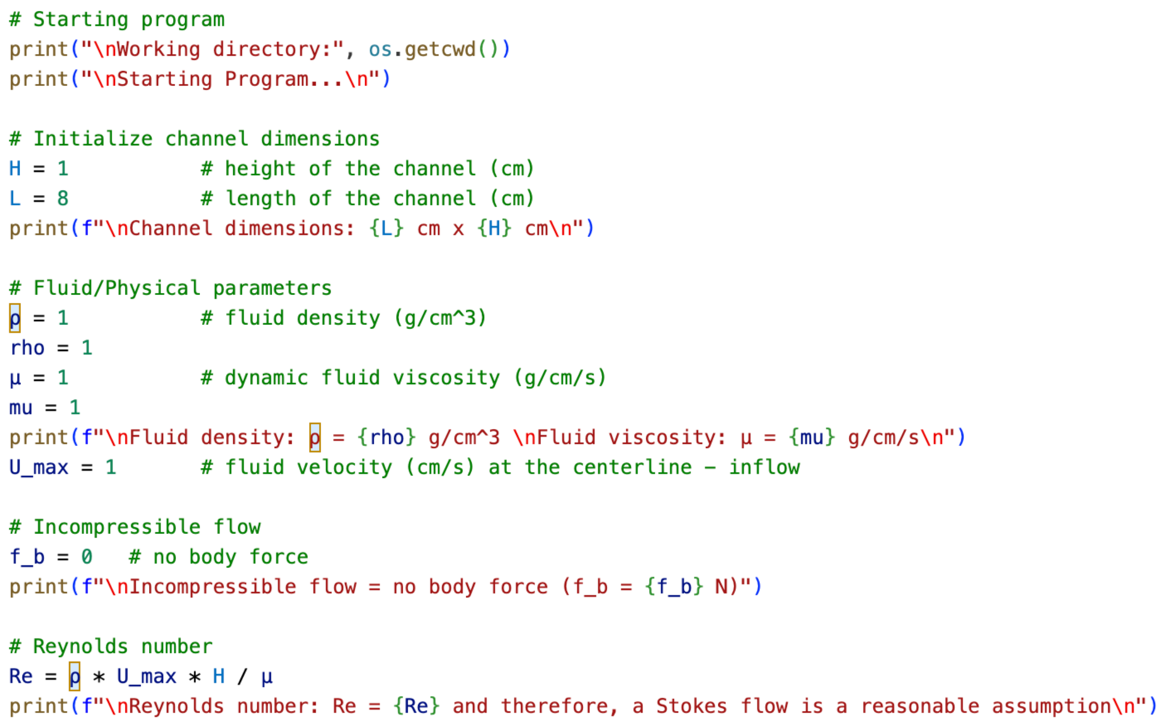
Simulating the problem using stabilized finite element (FE) method involves using a reasonably chosen grid and equal order discretization for fluid velocity and pressure. Examples include P1P1 (linear triangles for both velocity and pressure) and Q1Q1 (bilinear 4-noded quadrilaterals for both velocity and pressure). When solving the Stokes equations, a stabilization term (tau) is applied.

Simulating using *inf-sup* conforming finite elements is the same process as the FE method but without applying the PSPG (Pressure Stabilizing Petrov-Galerkin) stabilization parameter. An example is Q2Q1 discretization where the velocity functions are biquadratic (9-noded quadrilateral) and pressure is approximated using bilinear (4-noded quadrilateral) elements.

2. Methodology

The problem involves simulating incompressible 2D stokes flow in a rectangular channel using stabilized finite element methods (FEM) and *inf-sup* conforming finite elements. The stabilized approach (P1P1 or Q1Q1) enables equal-order discretization of velocity and pressure. A parabolic inflow velocity profile and Dirichlet boundary conditions are imposed. The P1P1 solution incorporates the Pressure Stabilizing Petrov-Galerkin (PSPG) stabilization term to ensure stability of the numerical scheme. The other method used is the *inf-sup* condition or Ladyzhenskaya–Babuška–Brezzi (LBB) condition, which is used for solving saddle-point problems in FEM, such as those encountered in the Stokes flow equations. It ensures stability and uniqueness of the solution, particularly for problems involving mixed formulations like velocity and pressure. An example of the *inf-sup* condition is Q2Q1 discretization. The Q2Q1 element pair is used in FEM configuration for solving incompressible flows like the Stokes equations. It combines higher-order velocity approximation with lower-order pressure approximation, which ensures stability and accuracy while satisfying the *inf-sup* condition.

2.1 Initializing parameters for P1P1 and Q2Q1



```
# Starting program
print("\nWorking directory:", os.getcwd())
print("\nStarting Program...\n")

# Initialize channel dimensions
H = 1           # height of the channel (cm)
L = 8           # length of the channel (cm)
print(f"\nChannel dimensions: {L} cm x {H} cm\n")

# Fluid/Physical parameters
ρ = 1           # fluid density (g/cm^3)
rho = 1
μ = 1           # dynamic fluid viscosity (g/cm/s)
mu = 1
print(f"\nFluid density: ρ = {rho} g/cm^3 \nFluid viscosity: μ = {mu} g/cm/s\n")
U_max = 1       # fluid velocity (cm/s) at the centerline - inflow

# Incompressible flow
f_b = 0   # no body force
print(f"\nIncompressible flow = no body force (f_b = {f_b} N)")

# Reynolds number
Re = ρ * U_max * H / μ
print(f"\nReynolds number: Re = {Re} and therefore, a Stokes flow is a reasonable assumption\n")
```

Figure 2: Initial parameters in the code for the simulations

The code begins by initializing the parameters mentioned in section one, **Objective**.

2.2 P1P1 Simulation

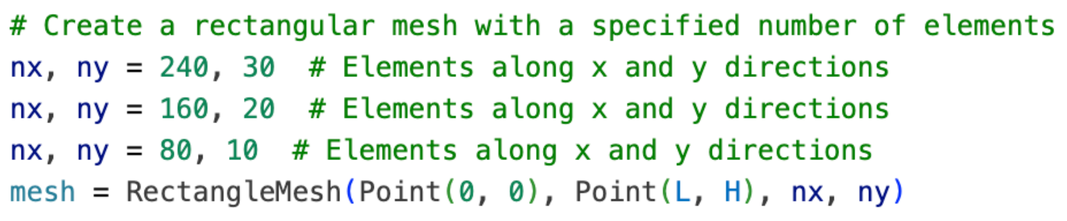
After the parameters have been initialized, the code begins the P1P1 simulation. P1P1 discretization is used for solving partial differential equations (PDEs) with the finite element method (FEM). Specifically, it describes the pairing of finite element spaces for different variables in a problem, such as velocity and pressure in incompressible flow problems (Stokes equations). The P1 refers to linear Lagrange elements (polynomial of degree 1). Thus, the solution (velocity or pressure) is approximated using piecewise linear functions over each element of the mesh. These elements are continuous, meaning that the solution is continuous across the boundaries of elements. A P1 element has one degree of freedom (DOF) per vertex (node) of the mesh. P1P1 refers to using P1 linear elements for both velocity and pressure. This pairing means that both the velocity field (**u**) and the pressure field (p) are approximated with piecewise linear basis functions over the same mesh, which makes P1P1 computationally simple. When solving P1P1's Stokes equations, the PSPG stabilization term is added.

2.3 Q2Q1 Simulation

The Q2Q1 simulation begins after the P1P1 simulation and follows the same steps except it solves different Stokes equations (no stabilization term) and has different function spaces. The Q2 represents the quadratic elements used for the velocity. The velocity field **u** is approximated using biquadratic shape functions (Q2). These shape functions are defined on a quadrilateral mesh and are second-order polynomials. Each element has nine velocity degrees of freedom (DOFs) per dimension (1 DOF at each corner node, 1 at each edge midpoint, and 1 in the center). The Q1 represents the linear elements for pressure. The pressure field p is approximated using bilinear shape functions (Q1). These are first-order polynomials defined over the same quadrilateral elements. Each element has 1 pressure degree of freedom at the center (or equivalently at the corner nodes if interpolated linearly). The Q2Q1 pairing satisfies the *inf-sup* condition for mixed velocity-pressure formulations making it stable and accurate for incompressible flow problems.

2.4 Mesh and Function Spaces

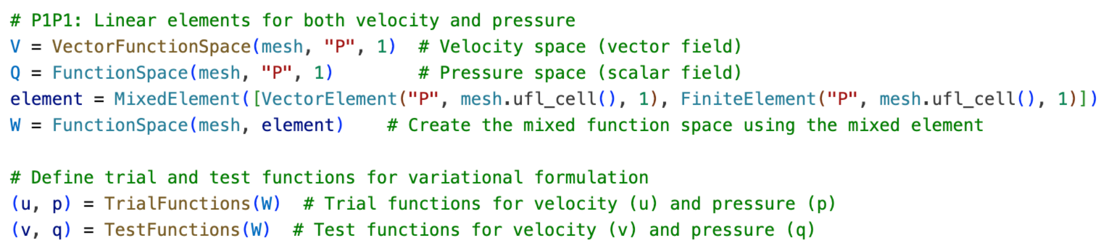
The first step in the simulation is to create the FEM mesh using the FEniCS library. The computational domain is discretized into a rectangular mesh of x-direction elements and y-direction elements. Three different mesh sizes were used to evaluate if the solutions improved from using a coarse mesh to a refined mesh.



```
# Create a rectangular mesh with a specified number of elements
nx, ny = 240, 30  # Elements along x and y directions
nx, ny = 160, 20  # Elements along x and y directions
nx, ny = 80, 10  # Elements along x and y directions
mesh = RectangleMesh(Point(0, 0), Point(L, H), nx, ny)
```

Figure 3: Code used to create FEM mesh in the code

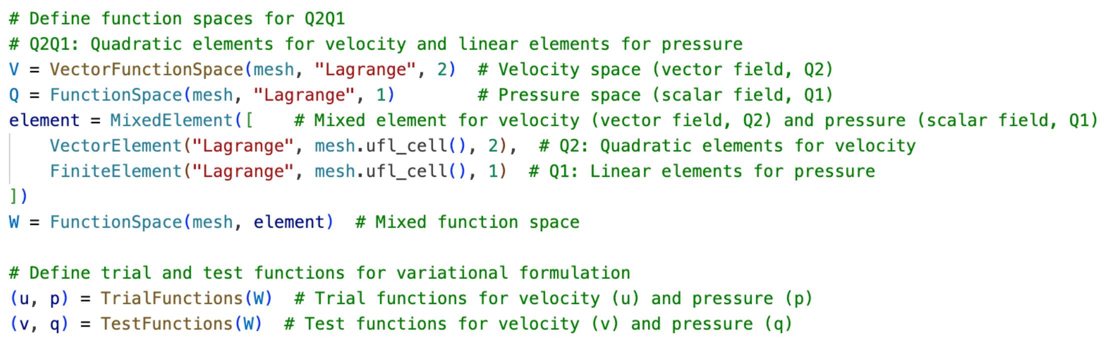
Separate function spaces are defined for velocity (as a vector field) and pressure (as a scalar field), both using linear elements in P1P1. For the Q2Q1 simulation, the function spaces are Lagrange, and the velocity elements are quadratic while the pressure elements are linear. To handle the coupled Stokes equations, a mixed function space is constructed using the MixedElement class in the FEniCS library, which combines the velocity and pressure spaces. The domain is initialized for a numerical solution by defining trial (u, p) and test (v, q) functions, which represent the velocity and pressure fields. Figure 4 shows the function spaces and trial/test functions defined for P1P1, and Figure 5 shows the same for Q2Q1. The main difference is the switch from a polynomial of degree 1 to a LaGrange polynomial of degree 2 for the velocity space/field.



```python
# P1P1: Linear elements for both velocity and pressure
V = VectorFunctionSpace(mesh, "P", 1)  # Velocity space (vector field)
Q = FunctionSpace(mesh, "P", 1)        # Pressure space (scalar field)
element = MixedElement([VectorElement("P", mesh.ufl_cell(), 1), FiniteElement("P", mesh.ufl_cell(), 1)])
W = FunctionSpace(mesh, element)    # Create the mixed function space using the mixed element

# Define trial and test functions for variational formulation
(u, p) = TrialFunctions(W)  # Trial functions for velocity (u) and pressure (p)
(v, q) = TestFunctions(W)  # Test functions for velocity (v) and pressure (q)
```

Figure 4: Function spaces and trial/test functions for P1P1 in the code



```python
# Define function spaces for Q2Q1
# Q2Q1: Quadratic elements for velocity and linear elements for pressure
V = VectorFunctionSpace(mesh, "Lagrange", 2)  # Velocity space (vector field, Q2)
Q = FunctionSpace(mesh, "Lagrange", 1)        # Pressure space (scalar field, Q1)
element = MixedElement([    # Mixed element for velocity (vector field, Q2) and pressure (scalar field, Q1)
    VectorElement("Lagrange", mesh.ufl_cell(), 2),  # Q2: Quadratic elements for velocity
    FiniteElement("Lagrange", mesh.ufl_cell(), 1)  # Q1: Linear elements for pressure
])
W = FunctionSpace(mesh, element)  # Mixed function space

# Define trial and test functions for variational formulation
(u, p) = TrialFunctions(W)  # Trial functions for velocity (u) and pressure (p)
(v, q) = TestFunctions(W)  # Test functions for velocity (v) and pressure (q)
```

Figure 5: Function spaces and trial/test functions for Q2Q1 in the code

2.5 Inflow Velocity and Analytical Solutions

The inflow velocity profile is specified analytically as a parabolic function, while analytical solutions for velocity and pressure gradients are computed for verification purposes.

The inflow velocity profile is calculated using the formula:

$$\frac{u_x}{U_\infty} = \frac{y}{H}(1 - \frac{y}{H})$$

$$u_x = u_{inlet} = U_\infty\left(\frac{y}{H}\left(1 - \frac{y}{H}\right)\right) = U_\infty\left(\frac{y}{H} - \frac{y^2}{H^2}\right) = U_\infty\left(\frac{yH - y^2}{H^2}\right) = \frac{U_\infty y(H - y)}{H^2}$$

$u_y = 0$

The analytical solution for the velocity gradient is computed using the formula:

$$u_x = \frac{U_\infty y(H - y)}{H^2} = \frac{(1)(y)(1 - y)}{(1)^2} = y(1 - y)$$

$$u_y = 0$$

where the y-values are the y-coordinate points from the mesh.

The analytical solution for the pressure gradient is calculated using the formula:

$$\frac{dp}{dx} = \frac{-2\mu U_\infty}{H^2} = \frac{-2(1)(1)}{(1)^2} = -2$$

The analytical solution for the pressure gradient will be the slope of the pressure curve. Using the formula $p = p_o - 2L$, the pressure can easily be calculated. If the initial pressure is 16psi, then the final pressure will be 0psi since

$$p = p_o - 2L = 16 - 2(8) = 0\text{psi}$$

2.6 Variational/Weak Formulation (Stokes Equations)

The Stokes equations, which govern steady, incompressible flows at low Reynolds numbers, are formulated in their weak form. The variational form for the Stokes system includes the viscous term, the pressure-velocity coupling term, and the continuity equation term. The three terms together formulate the bilinear form:

$$a_{Stokes} = \mu \int \nabla u : \nabla v \, dx - \int (\nabla \cdot v) p \, dx + \int (\nabla \cdot u) q \, dx$$

where u and p are the trial functions (velocity and pressure fields), v and q are the test functions, and μ is the viscosity. When solving Q2Q1 discretization, this equation is solved to obtain the bilinear form. The first term is the viscous term which represents the viscous forces in the momentum equation and accounts for the diffusion of momentum due to viscosity (μ). The second term is the pressure-velocity coupling term. This term enforces the incompressibility constraint (continuity equation) in the weak form and ensures that pressure is consistent with the divergence of test function v. The third term is the continuity equation term, which ensures that the velocity field u satisfies the incompressibility condition (divergence-free velocity field).

The linear forcing form is defined as:

$\nabla \cdot \boldsymbol{u} = 0$ (incompressibility)

$$L_{stokes} = \int \boldsymbol{f^b} \cdot v \, dx = 0$$

where $\boldsymbol{f^b}$ is the body force, which is zero, and v is the test function for the velocity. This equation is solved in Q2Q1 as the linear form. These equations are used in both P1P1 and Q2Q1, however, in P1P1 there is an added stabilization term (PSPG tau). To allow for equal-order discretization (P1P1 elements), the Pressure-Stabilizing Petrov-Galerkin (PSPG) method is incorporated. The stabilization term involves an additional penalty proportional to the pressure gradient, ensuring numerical stability:

$\tau = \frac{h^2}{4\mu}$, where h is the element diameter

Stabilization term: $\tau \int \nabla p \cdot \nabla q \, dx$

The stabilization term ensures the incompressibility condition and allows stable solutions with equal-order interpolation for velocity and pressure. In conclusion, when simulating P1P1, the stabilization term is added to the stokes equations. For the Q2Q1 simulation, the stokes equations above are used without the stabilization term.

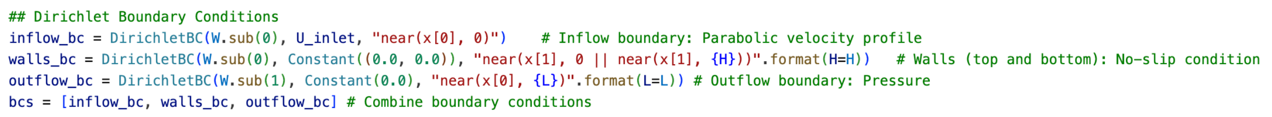
2.7 Boundary Conditions

The Dirichlet boundary conditions enforce physical constraints and allow the solution to match the expected behavior at the domain boundaries. The three Dirichlet boundary conditions are

- **Inflow boundary** (at x=0) with the parabolic velocity profile:

$$\frac{u}{U_\infty} = \frac{y}{H}\left(1 - \frac{y}{H}\right) \rightarrow u_{inlet} = \frac{U_\infty y(H-y)}{H^2}.$$

- **No-slip/penetration walls** (at y=0 and y=H) where the velocity is zero (**u** = 0, 0).
- **Outflow boundary** (at x = L) traction-free boundary condition where the pressure is set to zero: $t_{\hat{n}} = \boldsymbol{T} \cdot \hat{n} = 0$

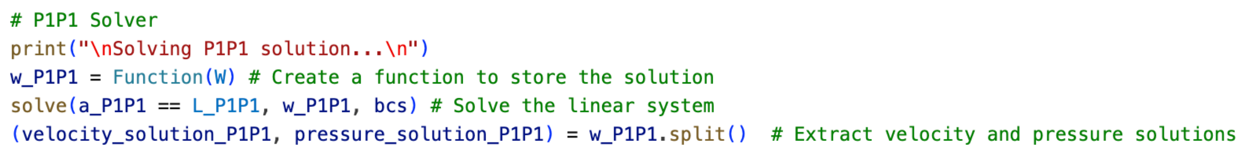


```
## Dirichlet Boundary Conditions
inflow_bc = DirichletBC(W.sub(0), U_inlet, "near(x[0], 0)")    # Inflow boundary: Parabolic velocity profile
walls_bc = DirichletBC(W.sub(0), Constant((0.0, 0.0)), "near(x[1], 0 || near(x[1], {H}))".format(H=H))   # Walls (top and bottom): No-slip condition
outflow_bc = DirichletBC(W.sub(1), Constant(0.0), "near(x[0], {L})".format(L=L)) # Outflow boundary: Pressure
bcs = [inflow_bc, walls_bc, outflow_bc] # Combine boundary conditions
```

Figure 6: Dirichlet Boundary Conditions for P1P1 and Q2Q1 in the code

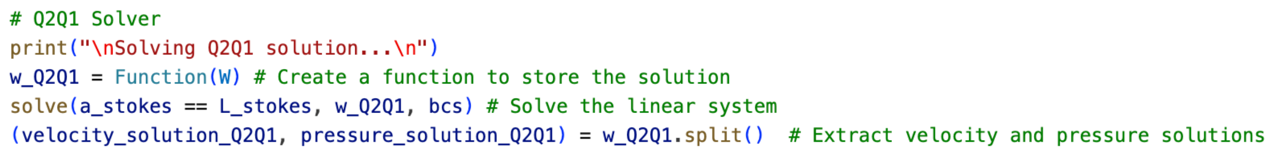
2.8 Numerical Solutions

P1P1, the stabilized system, and Q2Q1 are solved for velocity u and pressure p using the FEniCS solve function. The stabilized stokes functions are used for P1P1 and the regular bilinear and linear stokes equations are used for Q2Q1. The solutions are stored in a mixed finite element space W and then split into their respective velocity and pressure components.



```
# P1P1 Solver
print("\nSolving P1P1 solution...\n")
w_P1P1 = Function(W) # Create a function to store the solution
solve(a_P1P1 == L_P1P1, w_P1P1, bcs) # Solve the linear system
(velocity_solution_P1P1, pressure_solution_P1P1) = w_P1P1.split()  # Extract velocity and pressure solutions
```

Figure 7: Solving P1P1 with stabilized stokes equations and extracting the velocity and pressure solutions in the code

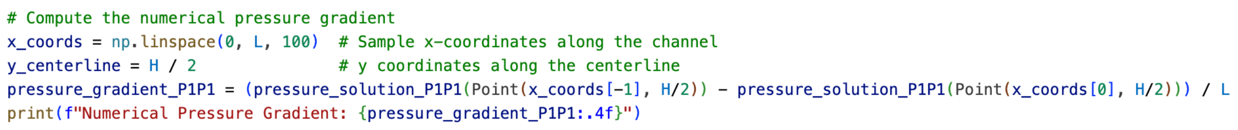


```
# Q2Q1 Solver
print("\nSolving Q2Q1 solution...\n")
w_Q2Q1 = Function(W) # Create a function to store the solution
solve(a_stokes == L_stokes, w_Q2Q1, bcs) # Solve the linear system
(velocity_solution_Q2Q1, pressure_solution_Q2Q1) = w_Q2Q1.split()  # Extract velocity and pressure solutions
```

Figure 8: Solving Q2Q1 with un-stabilized stokes equations and extracting the velocity and pressure solutions in the code

After the velocity and pressure solutions are extracted, the numerical pressure gradients are calculated for both P1P1 and Q2Q1. The pressure gradient formula used in the code is the same for P1P1 and Q2Q1:

$$pressure\ gradient = \frac{rightmost\ solution - leftmost\ solution}{Length\ of\ channel}$$



```python
# Compute the numerical pressure gradient
x_coords = np.linspace(0, L, 100)  # Sample x-coordinates along the channel
y_centerline = H / 2               # y coordinates along the centerline
pressure_gradient_P1P1 = (pressure_solution_P1P1(Point(x_coords[-1], H/2)) - pressure_solution_P1P1(Point(x_coords[0], H/2))) / L
print(f"Numerical Pressure Gradient: {pressure_gradient_P1P1:.4f}")
```

Figure 9: Computing the numerical pressure gradient in the code

The numerical pressure gradient along the centerline of the channel is compared against the analytical pressure gradient which results in the relative error. The relative error between the numerical and analytical pressure gradients is computed to assess accuracy.

$$relative\ error = \frac{pressure\ solution - analytical\ pressure\ gradient}{analytical\ pressure\ gradient} \cdot 100\%$$

The relative error between the numerical and analytical velocity gradients is calculated using the error norm function or the L2-Norm. The L2-Norm of the error measures the difference between the two velocity fields over the entire domain:

$$Error\ Norm = \sqrt{\int \left\| u_{analytical} - u_{numerical} \right\|^2}$$

This norm provides a quantitative measure of the difference between the numerical and analytical solutions. The relative error is computed by normalizing the error norm with respect to the L2-Norm of the analytical solution. The computed error is multiplied by 100 to convert it into a percentage, giving a relative error in terms of a percentage difference between the analytical and numerical velocity fields.

$$Norm\ of\ Analytical\ Solution = \sqrt{\int \left\| u_{analytical} \right\|^2}$$

$$Relative\ Error\ (\%) = \frac{Error\ Norm}{Norm\ of\ Analytical\ Solution} \cdot 100\%$$

3. Results

3.1 P1P1 Results

The results of the P1P1 simulation were successful. The calculated analytical pressure gradient for the problem is -2 from the formula previously mentioned $\left(\frac{dp}{dx} = \frac{-2\mu U_\infty}{H^2} = \frac{-2(1)(1)}{(1)^2} = -2\right)$. After the simulation was completed, the calculated numerical pressure gradient was -1.8719 for a coarse mesh size of 80x10, which yielded a relative error of 6.4026%. The L2-norm error calculation of the corresponding velocity gradient was 3.3430%. For a more refined mesh size of 160x20, the calculated numerical pressure gradient was -1.9628. This pressure gradient returned a relative error of 1.8586%, and the L2-norm error percentage for the velocity gradient corresponding to the 160x20 mesh was 0.87%. The most refined mesh of 240x30 elements produced a numerical pressure gradient of -1.9815, which corresponded to a relative error of just 0.9268%. This refined mesh size calculated an L2-norm error velocity gradient of 0.3898%.

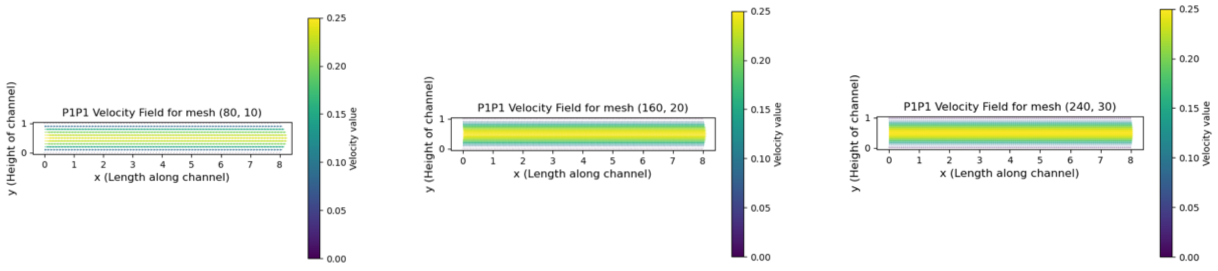


Figure 9: Velocity Fields for P1P1 at various mesh sizes

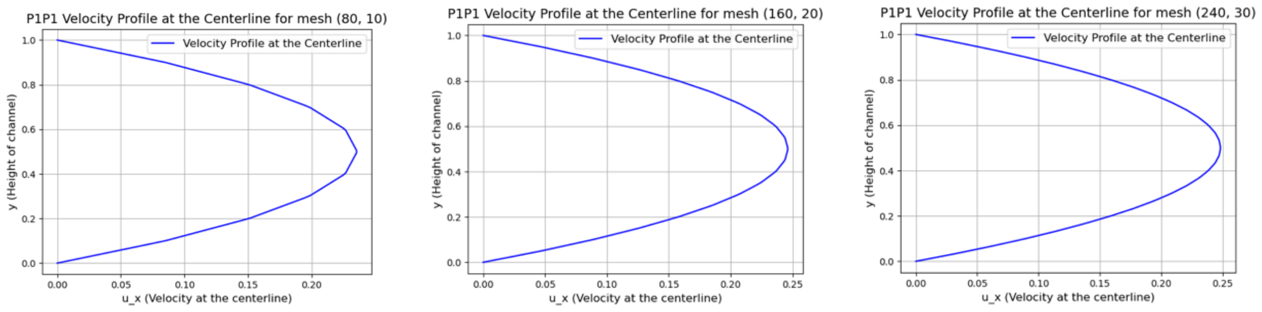


Figure 10: Velocity Profile at the Centerline for P1P1 at various mesh sizes

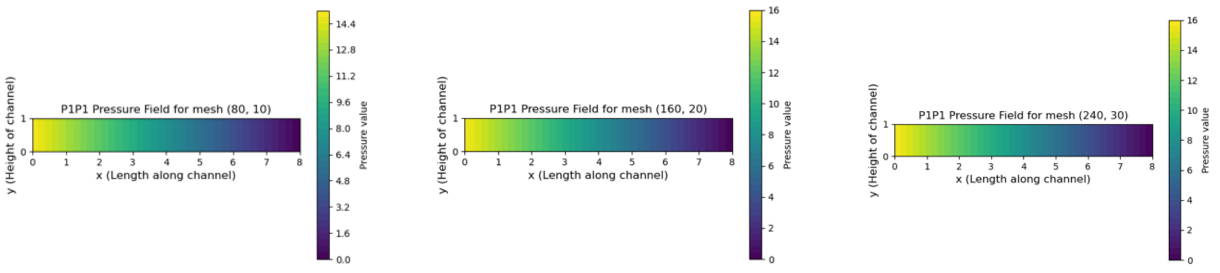


Figure 11: Pressure Field for P1P1 at various mesh sizes

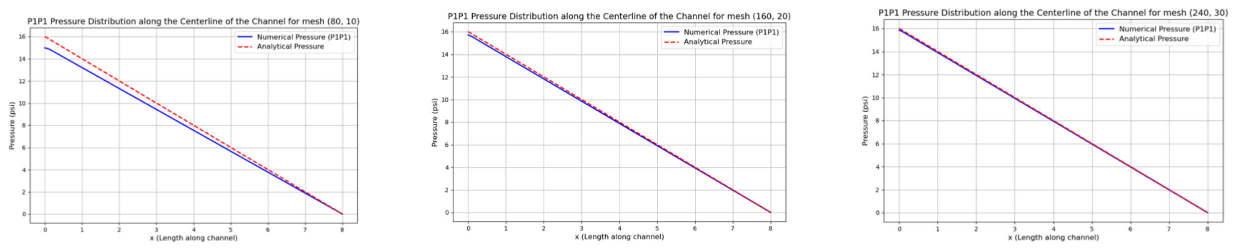


Figure 12: Pressure Distribution along the Centerline of the Channel for P1P1 at various mesh sizes

3.2 Q2Q1 Results

The results of the Q2Q1 simulation were very successful. The analytical pressure gradient of -2 was the same as the numerically calculated pressure gradient of -2.00, which yielded a relative error of 0.00%. The L2-norm error calculation of the velocity gradient was calculated as 0.00%. These results were consistent across all mesh sizes from the coarsest (80x10) to the most refined (240x30).

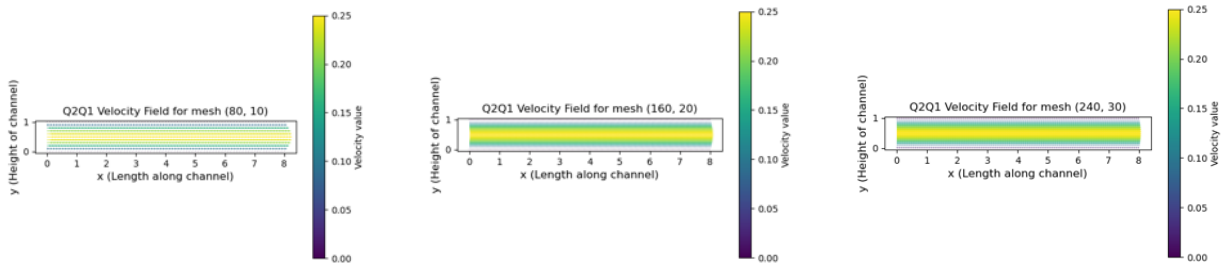


Figure 13: Q2Q1 Velocity Fields for various mesh sizes

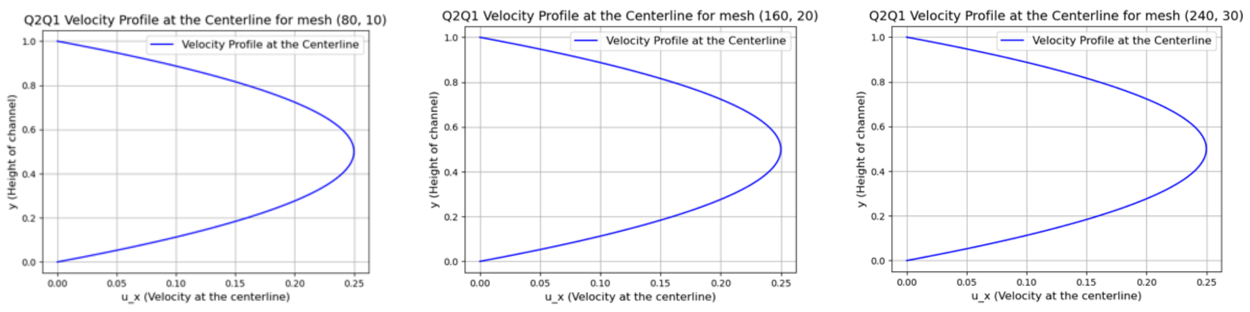


Figure 14: Q2Q1 Velocity Profile at the Centerline for various mesh sizes

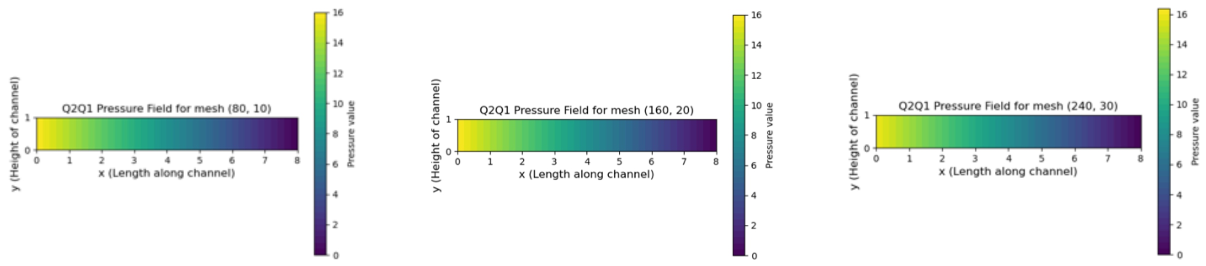


Figure 15: Q2Q1 Pressure Field for various mesh sizes

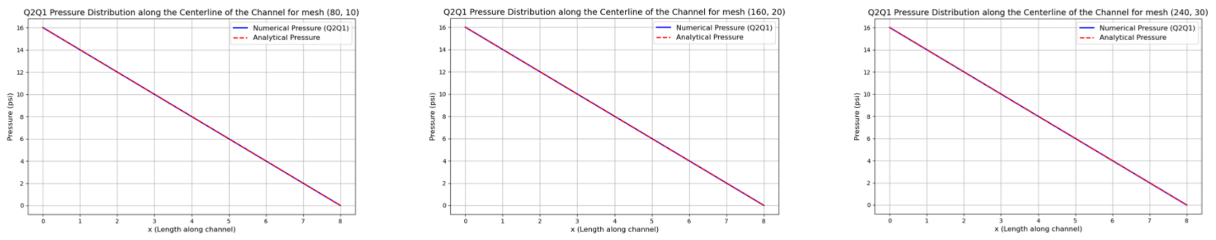


Figure 16: Q2Q1 Pressure Distribution along the Centerline of the Channel for various mesh sizes

4. Discussion

4.1 P1P1 Discussion

The P1P1 simulation gradually improved as the mesh size was refined. The three mesh sizes used were 80x10, 160x20, and 240x30. These represent the number of elements in the x and y directions within the 8cm x 1cm channel. For P1P1 discretization, there is a linear relationship between mesh size and improved accuracy. The relative error for the pressure gradient began at 6.4026%, went down to 3.3430%, and finished at 0.9268% for the most refined mesh size. Similarly, the relative L2-norm error for the velocity gradient began at 3.3430%, went down to 0.87% for the 160x20 mesh, and finished at 0.3898% for the 240x30 mesh.

Figure 10 shows the P1P1 velocity profiles at the centerline. As the mesh increased in size, the parabolic inflow became smother and less rigid. For the 80x10 mesh, the curve was linear, rigid and sharp near the vertex. As the mesh size increased these points became smooth and curved. The 240x30 mesh size produced an extremely smooth curve with almost no error.

Figure 12 shows the pressure distribution along the centerline of the channel. The same relationship follows for pressure; as the mesh size increased, the numerical solution became closer to the analytical solution. The slope of the curves is the pressure gradient, which represents the pressure drop over the course of the channel. The initial pressure is 16psi at the beginning of the channel (x=0) and the final pressure is 0psi at the end of the channel (x=8) from the pressure gradient formula, $p = p_o - 2L$. The analytical solution (slope) is -2, and these plots demonstrate that as the mesh size increases, the slope gets closer to the ideal solution.

4.2 Q2Q1 Discussion

The Q2Q1 simulation was perfect regardless of the mesh size. The relative error for the pressure and velocity gradients were consistently 0.00%. Figure 14 shows perfectly smooth parabolic inflow curves that have 0.00% error. Figure 16 shows the pressure distribution along the centerline of the channel that have a slope of -2.0 with a 0.00% error.

5. Conclusion

The simulations demonstrate a clear distinction in accuracy and performance between P1P1 and Q2Q1 discretization. For the P1P1 method, refinement of the mesh size significantly improved the accuracy of the results, with a notable reduction in relative errors for both pressure and velocity gradients as the mesh was refined from 80x10 to 240x30. The velocity profiles and pressure distributions increasingly aligned with the analytical solution, showcasing the expected parabolic inflow and linear pressure gradient. However, even at the most refined mesh size, the P1P1 discretization exhibited minor residual errors, indicating a limitation in its convergence rate.

In contrast, the Q2Q1 method achieved perfect results across all mesh sizes, with 0.00% relative errors for both pressure and velocity gradients. This underscores the superior accuracy and convergence properties of Q2Q1 discretization, which successfully captured the exact analytical solution for both velocity profiles and pressure distributions, regardless of mesh refinement.

Overall, while the P1P1 method benefits from mesh refinement, its accuracy is inherently constrained compared to the Q2Q1 method, which proves to be the optimal choice for simulating 2D Stokes flow in the channel. These results highlight the importance of selecting higher-order elements for achieving precise and reliable FEM simulations. In conclusion, the second order velocity polynomial combined with the first order pressure polynomial are a better fit for the problem than the first order polynomials for both pressure and velocity used in P1P1.

6. References

Dokken, Jørgen, Langtangen, Hans Peeter and Logg, Anders. *Test problem 1: Channel flow (Poiseuille flow).* The FEniCS Tutorial, 2023. https://jsdokken.com/dolfinx-tutorial/chapter2/ns_code1.html. Accessed 17 Dec. 2024.

Langtangen, Hans Petter and Pedersen, Geir K. *Advanced partial differential equation models*, pages 99–134. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-32726-6_4. Accessed 17 Dec. 2024.

Logg, Anders, Mardal, Kent-Andre and Wells, Garth N. *Automated Solution of Differential Equations by the Finite Element Metho*d. The FEniCS Project, 2011.

7. Appendix

```python
#/Users/nicolinoprimavera

# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import dolfin
import os
from fenics import *
import pandas as pd

# Check installation
print(f"\nChecking installation:")
#print(f"\nFEniCSx version (dolfinx): {dolfinx.__version__}")
#print(f"MPI initialized: {MPI.initialized()}")
#print(f"pyvista version: {pv.__version__}")
#print(f"UFL version: {ufl.__version__}")
print(f"FEniCS version (dolfin): {dolfin.__version__}")

"""Nicolino Primavera
FEM for Fluid Flow and FSI Interactions
Final Project
12/17/24

Problem II: Stokes Flow in a 2D Channel

Objective: compare the numerically predicted pressure gradient against anyalytical value assuming stokes flow

Stokes flow governing equations:
  - ρ(∂u/dt - f^b) = ∇ • σ
  - ∇ • u = 0 (incompressibility)
  - -∇p + μ∇^2u = 0     (∇^2u represents the Laplacian of velocity)

Stabilized Stokes Equations:
  - Petrov Galerkin method to enhance stability without losing consistency
  - Galerkin formulation + add stabilization terms
  - stabilization allows using equal-order discretization for both velocity and pressure function spaces (ex. P1-P1, Q1-Q1)

Variables:
  - σ_ij = -p*δ_ij + μ(u_i,j + u_j,i) - Cauchy stress for the fluid
  - u - fluid velocity
  - p - fluid pressure
  - f^b - body force acting on the fluid
  - ρ - fluid density
  - μ - dynamic viscosity

Channel dimensions: 1cm (height) x 8cm (length)

Boundary Conditions:
  - Inflow: Dirichlet condition with a velocity profile u/U_∞ = y/H * (1 - y/H) where U_∞ = 1 cm/s is the velocity at the centerline
  - Outflow: Traction-free boundary condition
  - Top/Bottom faces: Fixed walls (no slip / no penetration condition, u = 0)

Physical Parameters:
  - Incompressible flow, no body force (f^b = 0)
  - Fluid Density (ρ): 1.0 g/cm^3
  - Fluid Viscosity (μ): 1.0 g/cm/s

The Reynolds number of the flow Re = (ρ)*(U_∞)*(H)/μ = 1, therefore, a Stokes flow is a reasonable assumption for the above configuration

Problems:
- (a) Simulate the above problem using stabilized finite element (FE) method using a reasonably chosen grid and equal order discretization for fluid velocity and pressure. E.g., P1P1 (linear triangles for both velocity and pressure), Q1Q1 (bilinear 4-noded quadrilaterals for both velocity and pressure, etc.)
- (b) Simulate the above problem using inf-sup conforming finite elements. E.g., Q2Q1 discretization where the velocity functions are biquadratic (9-noded quadrilateral) and pressure is approximated using bilinear (4-noded quadrilateral) elements.
- (c) Plot profiles of velocity and streamlines at any axial cross-section.
- (d) Plot pressure along the centerline and compare the gradient against analytical expression assuming a fully developed parallel flow.
"""

# Starting program
print("\nWorking directory:", os.getcwd())
print("\nStarting Program...\n")

# Initialize channel dimensions
H = 1           # height of the channel (cm)
L = 8           # length of the channel (cm)
print(f"\nChannel dimensions: {L} cm x {H} cm\n")

# Fluid/Physical parameters
ρ = 1           # fluid density (g/cm^3)
rho = 1
μ = 1           # dynamic fluid viscosity (g/cm/s)
mu = 1
print(f"\nFluid density: ρ = {rho} g/cm^3 \nFluid viscosity: μ = {mu} g/cm/s\n")
U_max = 1       # fluid velocity (cm/s) at the centerline - inflow

# Incompressible flow
f_b = 0   # no body force
print(f"\nIncompressible flow = no body force (f_b = {f_b} N)")

# Reynolds number
Re = ρ * U_max * H / μ
print(f"\nReynolds number: Re = {Re} and therefore, a Stokes flow is a reasonable assumption\n")

# P1P1 Simulation with Stabilization
```

```python
print("\nSolving part (a): Running simulation with P1P1 linear elements (equal-order)...")

"""Part (a)
- Simulate the above problem using stabilized finite element (FE) method using a reasonably chosen grid and equal order discretization for fluid velocity and pressure.
- Ex: P1P1 (linear triangles for both velocity and pressure), Q1Q1 (bilinear 4-noded quadrilaterals for both velocity and pressure, etc.)
- Parts (c) and (d) plots are included
"""

# Create a rectangular mesh with a specified number of elements - CHANGE COMMENTS DEPENDING ON WHICH MESH SIZE YOU ARE SOLVING
#nx, ny = 240, 30  # Elements along x and y directions
#nx, ny = 160, 20  # Elements along x and y directions
nx, ny = 80, 10  # Elements along x and y directions
mesh = RectangleMesh(Point(0, 0), Point(L, H), nx, ny)

print("\nFEM mesh created.")

# P1P1: Linear elements for both velocity and pressure
V = VectorFunctionSpace(mesh, "P", 1)  # Velocity space (vector field)
Q = FunctionSpace(mesh, "P", 1)        # Pressure space (scalar field)
element = MixedElement([VectorElement("P", mesh.ufl_cell(), 1), FiniteElement("P", mesh.ufl_cell(), 1)]) # Define mixed element for velocity (vector field) and pressure (scalar field)
W = FunctionSpace(mesh, element)    # Create the mixed function space using the mixed element

# Define trial and test functions for variational formulation
(u, p) = TrialFunctions(W)  # Trial functions for velocity (u) and pressure (p)
(v, q) = TestFunctions(W)  # Test functions for velocity (v) and pressure (q)

# If you want to test W function space
w_test = Function(W)    # Create a zero function in the mixed space
u_mixed, p_mixed = w_test.split()   # Split into velocity and pressure components

#print("Creation of function spaces complete.\n")
print("\nVelocity function space is a vector field (2D).")
print("Pressure function space is a scalar field (1D).\n")
#print("Trial and test functions created.")

# Define the parabolic inflow velocity profile
U_max = 1.0 # Maximum velocity at the centerline
U_inlet = Expression(("U_max * x[1] * (H - x[1]) / pow(H, 2)", "0.0"), U_max=U_max, H=H, degree=2)  # U_max = centerline velocity
print(f"\nInflow velocity profile (analytical solution): {U_inlet}")

# Analytical solution
# Create a mesh grid for the whole domain
x_vals = np.linspace(0, L, 100) # L = 8
y_vals = np.linspace(0, H, 100) # H = 1
X, Y = np.meshgrid(x_vals, y_vals)
#print(f"\ny-values {y_vals}")
```

```

# Compute the velocity components across the domain
U_x = U_max * Y * (H - Y) / (H**2)  # Parabolic velocity profile in x-direction
U_y = np.zeros_like(Y)              # Zero velocity in the y-direction
print("\nAnalytical solution for inflow velocity profile:\n")
print(f"Parabolic inflow velocity in the x-direction:\n \n{U_x}\n")
print(f"Parabolic inflow velocity in the y-direction:\n \n{U_y}\n")

## Stokes Equations (weak/variational form) - for incompressible flow in a 2D domain

# Define the bilinear form for Stokes equations
a_stokes = (
    mu * inner(grad(u), grad(v)) * dx    # Viscous term - represents the viscous forces in the momentum eqn. - acounts for the diffusion of momentum due to viscosity(mu)
    - div(v) * p * dx                    # Pressure-velocity coupling term - enforces the incompressibility constraint (continuity eqn.) in the weak form - ensures pressure is consistent w/ divergence of test function v
    + div(u) * q * dx                    # Continuiy eqn. term - ensures that the velocity field u satisfies the incompressibility condition (divergence-free velocity field)
)
print(f"\nBilinear form of stokes equations:\n {a_stokes}")
# Define the linear form (forcing term)
f_b = Constant((0.0, 0.0))  # Zero body force for 2D flow
L_stokes = dot(f_b, v) * dx  # linear form , ∇ • u = 0 (incompressibility)
#L_stokes = dot(Constant((0.0, 0.0)), v) * dx  # linear form , ∇ • u = 0 (incompressibility)
print(f"\nLinear form of stokes equations:\n {L_stokes}\n")

# PSPG (Pressure Stabilizing Petrov-Galerkin) stabilization parameter τ (tau)
h = CellDiameter(mesh)  # Define characteristic element size (h) using UFL's CellDiameter
tau = h**2 / (4.0 * mu) # stabilization parameter
stabilization = tau * inner(grad(p), grad(q)) * dx  # stabilization term
print(f"\nPSPG stabilization parameter tau: {tau}")
print(f"Stabilization: {stabilization}\n")

# Stokes equations (stabilized) - stabilization allows using equal-order discretization for both velocity and pressure function spaces (ex. P1-P1, Q1-Q1)
a_P1P1 = (    # bilinear form
    mu * inner(grad(u), grad(v)) * dx    # Viscous term - represents the viscous forces in the momentum eqn. - acounts for the diffusion of momentum due to viscosity(mu)
    - div(v) * p * dx                    # Pressure-velocity coupling term - enforces the incompressibility constraint (continuity eqn.) in the weak form - ensures pressure is consistent w/ divergence of test function v
    + div(u) * q * dx                    # Continuiy eqn. term - ensures that the velocity field u satisfies the incompressibility condition (divergence-free velocity field)
    + tau * inner(grad(p), grad(q)) * dx # PSPG stabilization term
)
print(f"\nBilinear form of stokes equations with stabilization term: \n{a_P1P1}")
```

```python
L_P1P1 = dot(f_b, v) * dx  # linear form , ∇ • u = 0 (incompressibility)
#L_P1P1 = (dot(f_b, v) * dx
        # + tau * inner(grad(p), grad(q)) * dx
#) - added later on to keep architecture of L_P1P1 in tact
print(f"\nLinear form of stokes equations with stabilization term: \n{L_P1P1}")

## Dirichlet Boundary Conditions
inflow_bc = DirichletBC(W.sub(0), U_inlet, "near(x[0], 0)")    # Inflow boundary: Parabolic velocity profile
walls_bc = DirichletBC(W.sub(0), Constant((0.0, 0.0)), "near(x[1], 0 || near(x[1], {H}))".format(H=H))   # Walls (top and bottom): No-slip condition
outflow_bc = DirichletBC(W.sub(1), Constant(0.0), "near(x[0], {L})".format(L=L)) # Outflow boundary: Pressure
bcs = [inflow_bc, walls_bc, outflow_bc] # Combine boundary conditions

# Error handling
#print(f"Inflow boundary condition: \n{bcu_inflow}\n")
#print(f"Walls boundary condition: \n{bcu_walls}\n")
#print(f"Combined boundary conditions: \n{bcs}\n")

print("\nDirichlet boundary conditions defined.")

# P1P1 Solver
print("\nSolving P1P1 solution...\n")
w_P1P1 = Function(W) # Create a function to store the solution
solve(a_P1P1 == L_P1P1, w_P1P1, bcs) # Solve the linear system
(velocity_solution_P1P1, pressure_solution_P1P1) = w_P1P1.split()  # Extract velocity and pressure solutions

# P1P1 Velocity solutions
velocity_P1P1 = velocity_solution_P1P1.compute_vertex_values(mesh)
#print(f"\nVelocity: \n{velocity_P1P1}")    # error handling
if not np.isfinite(velocity_P1P1).all():
    raise ValueError("\nVelocity contains non-finite values (NaN or Inf).")

# P1P1 Pressure solutions
pressure_P1P1 = pressure_solution_P1P1.compute_vertex_values(mesh)
#print(f"\nPressure: \n{pressure_P1P1}\n")    # error handling
if not np.isfinite(pressure_P1P1).all():
    raise ValueError("\nPressure contains non-finite values (NaN or Inf).")

## P1P1 - Compute the numerical pressure gradient along the centerline
print("\nThe pressure gradient provides the slope of the pressure curve.")
# Pressure - example of an analytical solution
p_o = 16   # initial pressure
p_f = 0 # final pressure (analytical solution)
print(f"Pressure Gradient Analytical Solution:\n    - If the initial pressure: p_o = {p_o}psi, then the final pressure: p_f = {p_f}psi")
print(f"    - This is because of the formula:  p = p_o - 2 * L\n")
# Compute analytical pressure gradient: dp/dx = -2 * μ * U_max / (H**2)
analytical_pressure_gradient = -2 * μ * U_max / (H**2)
print(f"Analytical Pressure Gradient = dp/dx = -2 * μ * U_max / (H**2) = {analytical_pressure_gradient}")
# Compute the numerical pressure gradient
```

```python
    x_coords = np.linspace(0, L, 100)  # Sample x-coordinates along the channel
    y_centerline = H / 2               # y coordinates along the centerline
    pressure_gradient_P1P1 = (pressure_solution_P1P1(Point(x_coords[-1], H/2)) - pressure_solution_P1P1(Point(x_coords[0], H/2))) / L
    print(f"Numerical Pressure Gradient: {pressure_gradient_P1P1:.4f}")

    # Relative error between numerical and analytical pressure gradients
    relative_error_P1P1 = abs((pressure_gradient_P1P1 - analytical_pressure_gradient) / analytical_pressure_gradient) * 100
    print(f"Relative Error: {relative_error_P1P1:.4f}%\n")

    # Relative error between numerical and analytical velocity gradients
    error = errornorm(U_inlet, velocity_solution_P1P1, norm_type="L2") * 100
    print(F"Relative error for velocity: {error:.4f}%\n")

    # Create folder to save plots
    save_folder = "/Users/nicolinoprimavera/Desktop/Columbia University/Finite Element Method for Fluid Flow and Fluid-Structure Interactions/Final Project/Plots"
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)

    # P1P1 - Plot velocity field (magnitude)
    print(f"\nVelocity solution plot: \n{velocity_solution_P1P1}")
    plt.figure()
    plot_u_P1P1 = plot(velocity_solution_P1P1, title=f"P1P1 Velocity Field for mesh ({nx}, {ny})", cmap=cm.viridis)  # Use a colormap
    plt.colorbar(plot_u_P1P1, label="Velocity value")  # Attach the colorbar to the mappable object
    plt.xlabel("x (Length along channel)", fontsize=12)
    plt.ylabel("y (Height of channel)", fontsize=12)
    # Save the plot
    filename = f"P1P1 Velocity Field for mesh {nx} x {ny}.png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")
    plt.show()

    # P1P1 - Velocity profile at centerline
    print(f"\nVelocity profile at the centerline:\n")
    y_coords = np.linspace(0, H, 100)  # Sample y-coordinates from bottom to top of the channel
    x_centerline = L / 2
    u_centerline = [velocity_solution_P1P1(Point(x_centerline, y))[0] for y in y_coords]  # extract the velocity values along the centerline
    # Plot
    plt.figure()
    plt.plot(u_centerline, y_coords, label=f"Velocity Profile at the Centerline", color='blue')
    plt.xlabel("u_x (Velocity at the centerline)", fontsize=12)
    plt.ylabel("y (Height of channel)", fontsize=12)
    plt.title(f"P1P1 Velocity Profile at the Centerline for mesh ({nx}, {ny})", fontsize=14)
    plt.legend(fontsize=12)
    plt.grid(True)
    # Save the plot
    filename = f"P1P1 Velocity Profile at the Centerline for mesh ({nx}, {ny}).png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")
    plt.show()

    # P1P1 - Plot pressure field
    print(f"\nPressure solution plot: \n{pressure_solution_P1P1}\n")
    plt.figure()
    plot_p_P1P1 = plot(pressure_solution_P1P1, title=f"P1P1 Pressure Field for mesh ({nx}, {ny})", cmap=cm.viridis)  # Use a colormap
    plt.colorbar(plot_p_P1P1, label="Pressure value")  # Attach the colorbar to the mappable object
    plt.xlabel("x (Length along channel)", fontsize=12)
    plt.ylabel("y (Height of channel)", fontsize=12)
    # Save the plot
    filename = f"P1P1 Pressure Field for mesh ({nx}, {ny}).png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")
    plt.show()

    ## Plot the pressure along the centerline
    print(f"Pressure along the centerline plot:\n")
    # Initialize arrays to store numerical and analytical pressure values
    numerical_pressure_P1P1 = []
    analytical_pressure_P1P1 = []

    # Compute pressure values along the centerline
    for x in x_coords:
        numerical_pressure_P1P1.append(pressure_solution_P1P1(Point(x, H/2)))  # Numerical pressure
        analytical_pressure_P1P1.append(p_o + analytical_pressure_gradient * x)  # Analytical pressure

    # Plot the pressure distributions
    plt.figure(figsize=(10, 6))
    plt.plot(x_coords, numerical_pressure_P1P1, label="Numerical Pressure (P1P1)", color="blue", linewidth=2)
    plt.plot(x_coords, analytical_pressure_P1P1, label="Analytical Pressure", color="red", linestyle="--", linewidth=2)
    plt.xlabel("x (Length along channel)", fontsize=12)
    plt.ylabel("Pressure (psi)", fontsize=12)
    plt.title(f"P1P1 Pressure Distribution along the Centerline of the Channel for mesh ({nx}, {ny})", fontsize=14)
    plt.grid(True)
    plt.legend(fontsize=12)
    # Save the plot
    filename = f"P1P1 Pressure Distribution along the Centerline of the Channel for mesh ({nx}, {ny}).png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")
```

```
plt.show()

"""Part b
- Simulate the above problem using inf-sup conforming finite elements.
- E.g., Q2Q1 discretization where the velocity functions are biquadratic (9-noded quadrilateral) and pressure is approximated using bilinear (4-noded quadrilateral) elements.
- Parts (c) and (d) plots are included
"""
print(f"\nSolving part (b): Running simulation with Q2Q1 elements. \nVelocity functions (Q2) are biquadratic elements (9-noded quadrilateral) and pressure functions (Q1) are bilinear elements (4-noded quadrilateral).")

# Mesh
#nx, ny = 80, 10  # Elements along x and y directions
#mesh = RectangleMesh(Point(0, 0), Point(L, H), nx, ny)

# Define function spaces for Q2Q1
# Q2Q1: Quadratic elements for velocity and linear elements for pressure
V = VectorFunctionSpace(mesh, "Lagrange", 2)  # Velocity space (vector field, Q2)
Q = FunctionSpace(mesh, "Lagrange", 1)        # Pressure space (scalar field, Q1)
element = MixedElement([    # Mixed element for velocity (vector field, Q2) and pressure (scalar field, Q1)
    VectorElement("Lagrange", mesh.ufl_cell(), 2),  # Q2: Quadratic elements for velocity
    FiniteElement("Lagrange", mesh.ufl_cell(), 1)  # Q1: Linear elements for pressure
])
W = FunctionSpace(mesh, element)  # Mixed function space

# Define trial and test functions for variational formulation
(u, p) = TrialFunctions(W)  # Trial functions for velocity (u) and pressure (p)
(v, q) = TestFunctions(W)  # Test functions for velocity (v) and pressure (q)

#print(f"Q2Q1 function spaces defined.\n")
print("\nVelocity function space (Q2) is a vector field (2D) with biquadratic elements (9-noded quadrilateral).")
print("Pressure function space is a scalar field (1D) with bilinear elements (4-noded quadrilateral).\n")

# Define the parabolic inflow velocity profile
U_max = 1
U_inlet = Expression(("U_max * x[1] * (H - x[1]) / pow(H, 2)", "0.0"), U_max=1.0, H=H, degree=2)  # U_max = centerline velocity
print(f"\nInflow velocity profile: {U_inlet}\n")

# Compute the velocity components across the domain
U_x = U_max * Y * (H - Y) / (H**2)  # Parabolic velocity profile in x-direction
U_y = np.zeros_like(Y)              # Zero velocity in the y-direction
print("\nAnalytical solution for inflow velocity:\n")
print(f"Parabolic inflow velocity in the x-direction:\n \n{U_x}\n")
print(f"Parabolic inflow velocity in the y-direction:\n \n{U_y}\n")

## Stokes Equations - weak/variational form
```

```

# Define the bilinear form for Stokes equations
a_stokes = (
    mu * inner(grad(u), grad(v)) * dx    # Viscous term - represents the viscous forces in the momentum eqn. - acounts for the diffusion of momentum due to viscosity(mu)
    - div(v) * p * dx                    # Pressure-velocity coupling term - enforces the incompressibility constraint (continuity eqn.) in the weak form - ensures pressure is consistent w/ divergence of test function v
    + div(u) * q * dx                    # Continuiy eqn. term - ensures that the velocity field u satisfies the incompressibility condition (divergence-free velocity field)
)
print(f"\nBilinear form of stokes equations:\n {a_stokes}")
# Define the linear form (forcing term)
f_b = Constant((0.0, 0.0))  # Zero body force for 2D flow
L_stokes = dot(f_b, v) * dx  # linear form , ∇ • u = 0 (incompressibility)
print(f"\nLinear form of stokes equations:\n {L_stokes}\n")

## Dirichlet Boundary Conditions
inflow_bc = DirichletBC(W.sub(0), U_inlet, "near(x[0], 0)")    # Inflow boundary: Parabolic velocity profile
walls_bc = DirichletBC(W.sub(0), Constant((0.0, 0.0)), "near(x[1], 0 || near(x[1], {H}))".format(H=H))   # Walls (top and bottom): No-slip condition
outflow_bc = DirichletBC(W.sub(1), Constant(0.0), "near(x[0], {L})".format(L=L)) # Outflow boundary: Pressure
bcs = [inflow_bc, walls_bc, outflow_bc] # Combine boundary conditions

print("\nDirichlet boundary conditions defined.")

# Q2Q1 Solver
print("\nSolving Q2Q1 solution...\n")
w_Q2Q1 = Function(W) # Create a function to store the solution
solve(a_stokes == L_stokes, w_Q2Q1, bcs) # Solve the linear system
(velocity_solution_Q2Q1, pressure_solution_Q2Q1) = w_Q2Q1.split()  # Extract velocity and pressure solutions

# Velocity solutions
velocity_Q2Q1 = velocity_solution_Q2Q1.compute_vertex_values(mesh)
#print(f"\nVelocity: \n{velocity_Q2Q1}")    # error handling
if not np.isfinite(velocity_Q2Q1).all():
    raise ValueError("\nVelocity contains non-finite values (NaN or Inf).\n")

# Pressure solutions
pressure_Q2Q1 = pressure_solution_Q2Q1.compute_vertex_values(mesh)
#print(f"\nPressure: \n{pressure_Q2Q1}\n")    # error handling
if not np.isfinite(pressure_Q2Q1).all():
    raise ValueError("\nPressure contains non-finite values (NaN or Inf).\n")

# Compute the numerical pressure gradient along the centerline
#print("\nThe pressure gradient provides the slope of the pressure curve.")
# Analytical pressure gradient - dp/dx = -2 * μ * U_max / (H**2)
print(f"Analytical Pressure Gradient = dp/dx = {analytical_pressure_gradient}") # solved in part a
```

```python
    x_coords_Q2Q1 = np.linspace(0, L, 100)  # Sample x-coordinates along the channel
    pressure_gradient_Q2Q1 = (pressure_solution_Q2Q1(Point(x_coords[-1], H/2)) - pressure_solution_Q2Q1(Point(x_coords[0], H/2))) / L
    print(f"Numerical Pressure Gradient: {pressure_gradient_Q2Q1:.4f}")

    # Relative error between numerical and analytical pressure gradients
    relative_error_Q2Q1 = abs((pressure_gradient_Q2Q1 - analytical_pressure_gradient) / analytical_pressure_gradient) * 100
    print(f"Relative Error: {relative_error_Q2Q1:.2f}%")

    # Relative error between numerical and analytical velocity gradients
    error = errornorm(U_inlet, velocity_solution_Q2Q1, norm_type="L2") * 100
    print(F"Relative error for velocity: {error:.4f}%\n")

    # Plot velocity field (magnitude)
    print(f"\nVelocity solution plot: \n{velocity_solution_Q2Q1}")
    plt.figure()
    plot_u = plot(velocity_solution_Q2Q1, title=f"Q2Q1 Velocity Field for mesh ({nx}, {ny})", cmap=cm.viridis)  # Use a colormap
    plt.colorbar(plot_u, label="Velocity value")  # Attach the colorbar to the mappable object
    plt.xlabel("x (Length along channel)", fontsize=12)
    plt.ylabel("y (Height of channel)", fontsize=12)
    # Save the plot
    filename = f"Q2Q1 Velocity Field for mesh ({nx}, {ny}).png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")
    plt.show()

    # Q2Q1 - Velocity profile at centerline
    print(f"\nVelocity profile at the centerline:\n")
    y_coords = np.linspace(0, H, 100)  # Sample y-coordinates from bottom to top of the channel
    x_centerline = L / 2
    u_centerline = [velocity_solution_Q2Q1(Point(x_centerline, y))[0] for y in y_coords]  # extract the velocity values along the centerline
    # Plot
    plt.figure()
    plt.plot(u_centerline, y_coords, label="Velocity Profile at the Centerline", color='blue')
    plt.xlabel("u_x (Velocity at the centerline)", fontsize=12)
    plt.ylabel("y (Height of channel)", fontsize=12)
    plt.title(f"Q2Q1 Velocity Profile at the Centerline for mesh ({nx}, {ny})", fontsize=14)
    plt.legend(fontsize=12)
    plt.grid(True)
    # Save the plot
    filename = f"Q2Q1 Velocity Profile at the Centerline for mesh ({nx}, {ny}).png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")
    plt.show()

```

```python
# Plot pressure field
print(f"\nPressure solution plot: \n{pressure_solution_Q2Q1}\n")
plt.figure()
plot_p = plot(pressure_solution_Q2Q1, title=f"Q2Q1 Pressure Field for mesh ({nx}, {ny})", cmap=cm.viridis)  # Use a colormap
plt.colorbar(plot_p, label="Pressure value")  # Attach the colorbar to the mappable object
plt.xlabel("x (Length along channel)", fontsize=12)
plt.ylabel("y (Height of channel)", fontsize=12)
# Save the plot
filename = f"Q2Q1 Pressure Field for mesh ({nx}, {ny}).png"
save_path = os.path.join(save_folder, filename)
plt.savefig(save_path)
print(f"\nGraph saved as {save_path}.\n")
plt.show()

## Plot the pressure along the centerline
print(f"Pressure along the centerline plot:\n")
# Initialize arrays to store numerical and analytical pressure values
numerical_pressure_Q2Q1 = []
analytical_pressure_Q2Q1 = []

# Compute pressure values along the centerline
for x in x_coords:
    numerical_pressure_Q2Q1.append(pressure_solution_Q2Q1(Point(x, H/2)))  # Numerical pressure
    analytical_pressure_Q2Q1.append(p_o + analytical_pressure_gradient * x)  # Analytical pressure

# Plot the pressure distributions
plt.figure(figsize=(10, 6))
plt.plot(x_coords, numerical_pressure_Q2Q1, label="Numerical Pressure (Q2Q1)", color="blue", linewidth=2)
plt.plot(x_coords, analytical_pressure_Q2Q1, label="Analytical Pressure", color="red", linestyle="--", linewidth=2)
plt.xlabel("x (Length along channel)", fontsize=12)
plt.ylabel("Pressure (psi)", fontsize=12)
plt.title(f"Q2Q1 Pressure Distribution along the Centerline of the Channel for mesh ({nx}, {ny})", fontsize=14)
plt.grid(True)
plt.legend(fontsize=12)
# Save the plot
filename = f"Q2Q1 Pressure Distribution along the Centerline of the Channel for mesh ({nx}, {ny}).png"
save_path = os.path.join(save_folder, filename)
plt.savefig(save_path)
print(f"Graph saved as {save_path}.\n")
plt.show()
```