Testing Plan - Team 1
Paul, Nina, Jeff & Kelly

- ConditionParser
  - onlyXEquals()
    - test if returned filter for “x = 0” accurately matches points for x = 0
  - onlyXGreater()
    - test if returned filter for “x > 0” accurately matches points for x > 0
  - onlyXLess()
    - test if returned filter for “x < 0” accurately matches points for x < 0
  - onlyYEquals()
    - test if returned filter for “y = 0” accurately matches points for y = 0
  - onlyYGreater()
    - test if returned filter for “y > 0” accurately matches points for y > 0
  - onlyYLess()
    - test if returned filter for “y < 0” accurately matches points for y < 0
  - XY()
    - test if returned filter for “x=0,y<0” accurately matches points for x = 0 and y < 0
  - YX()
    - test if returned filter for “y=0,x>0” accurately matches points for x > 0 and y = 0
  - Doubles()
    - test if returned filter for “x>0.0,y>0.0” accurately matches points for x > 0 and y > 0
  - HalfDoubles()
    - test if returned filter for “x>0.,y>0.” accurately matches points for x > 0 and y > 0
  - ManyArgs1()
    - test if returned filter for “y=0,x>0,y=3” accurately matches points for y = 0, x > 0, and y = 3
  - ManyArgs2()
    - test if returned filter for “x>0,x<2,y<2” accurately matches points for x > 0, x < 2, and y < 2
  - OutOfOrder()
    - test if “0=x” throws a ValueError
  - NoComma()
    - test if “x=0 y<2” throws a ValueError
- SolveFormulation
  - For all tests, form is the formulation being used with SolutionFns functions and foo is the formulation being used with PyCamellia functions

- solve()
    - For each kind of formulation listed below, solve a formulation once using solve() and once using PyCamellia functions, then confirm the results are the same. Do so by comparing energy error, element count, and global degrees of freedom.
        - Transient Stokes
        - Steady State Stokes
        - Steady State Navier Stokes

- SolutionFns
    - For all tests, form is the formulation being used with SolutionFns functions and foo is the formulation being used with PyCamellia functions
    - For each test, create and solve a formulation once using SolutionFns and once using PyCamellia functions, anything that is done to form by SolutionFns functions must be done to foo by PyCamellia functions. Then confirm the results are the same. Do so by comparing energy error, element count, and global degrees of freedom.
    - steadyLinearInit()
        - Create form using steadyLinearInit() and test as previously described.
    - addWall()
        - Create form using steadyLinearInit(). Using notTopBoundary, add a wall to form using addWall(). Test as previously described.
    - addInflow()
        - Create form using steadyLinearInit(). Using topBoundary and topVelocity, add an inflow condition to form using addInflow(). Test as previously described.
        - Then, using notTopBoundary, add a wall to form using addWall(). Test as previously described. Do this because it should change the value for energyError because of the inflow condition.
    - addOutflow()
        - Create a form using steadyLinearInit(). Add an outflow condition to form using addOutflow(). Test as previously described.
    - energyPerCell()
        - Create form using steadyLinearInit(). Add a wall and an inflow condition. Use energyPerCell() to get the list of energy error per cell of form. Confirm that for each cell, the energy error is the same as foo’s.
    - steadyLinearSolve()
        - Create form using steadyLinearInit(). Add a wall and an inflow condition. Solve using steadyLinearSolve() and test as previously described.
    - steadyLinearHAutoRefine()

        - Create form using steadyLinearInit(). Add a wall and an inflow condition. Refine form using steadyLinearHAutoRefine(). Test as previously described.
    - steadyLinearPAutoRefine()
        - Create form using steadyLinearInit(). Add a wall and an inflow condition. Refine form using steadyLinearPAutoRefine(). Test as previously described.
    - steadyLinearHManualRefine()
        - Create form using steadyLinearInit(). Add a wall and an inflow condition. Retrieve the cell IDs for all active cells. Use those cell IDs to refine form using steadyLinearHManualRefine(). Test as previously described.
    - steadyLinearPManualRefine()
        - Create form using steadyLinearInit(). Add a wall and an inflow condition. Retrieve the cell IDs for all active cells. Use those cell IDs to refine form using steadyLinearPManualRefine(). Test as previously described.
    - transientLinearInit()
        - Create form using transientLinearInit() and test as previously described.
    - transientLinearSolve()
        - Create form using transientLinearInit(). Add a wall and an inflow condition. Solve using transientLinearSolve() and test as previously described.
    - steadyNonlinearInit()
        - Create form using steadyNonlinearInit() and test as previously described.
    - nonlinearSolve()
        - This is a helper method, it’s main difference from steadyNonlinearSolve() is that it is much less verbose.
        - Create form using steadyNonlinearInit(). Add a wall and an inflow condition. Solve using nonlinearSolve() and test as previously described.
    - steadyNonlinearSolve()
        - Create form using steadyNonlinearInit(). Add a wall and an inflow condition. Solve using nonlinearSolve() and test as previously described.
    - steadyNonlinearHAutoRefine()
        - Create form using steadyNonlinearInit(). Add a wall and an inflow condition. Refine form using steadyNonlinearHAutoRefine(). Test as previously described.
    - steadyNonlinearPAutoRefine()
        - Create form using steadyNonlinearInit(). Add a wall and an inflow condition. Refine form using steadyNonlinearPAutoRefine(). Test as previously described.
    - steadyNonlinearHManualRefine()
        - Create form using steadyNonlinearInit(). Add a wall and an inflow condition. Retrieve the cell IDs for all active cells.

Use those cell IDs to refine form using steadyNonlinearHManualRefine(). Test as previously described.

  - steadyNonlinearPManualRefine()
    - Create form using steadyNonlinearInit(). Add a wall and an inflow condition. Retrieve the cell IDs for all active cells. Use those cell IDs to refine form using steadyNonlinearPManualRefine(). Test as previously described.
- InputData
  - Memento
    - get() & set()
      - Create an instance of InputData using stokes and create a memento. Call get() and confirm that the resulting dataList contains stokes and does not contain nStokes. Then call set() and give it nStokes. Call get() again and confirm that the resulting dataList now contains nStokes and not stokes.
  - InputData
    - init()
      - Create an instance of InputData using stokes and create a memento. Call get() and confirm that the resulting dataList contains stokes.
    - setForm() & getForm()
      - Create an instance of InputData using stokes. Call setForm() and give it a form. Confirm that the form resulting from calling getForm() is the same instance as the form given in setForm().
    - addVariable() & getVariable()
      - Create an instance of InputData using stokes and call addVariable() with transient. call getVariable() for transient and stokes to confirm that they are equal.
    - createMemento()
      - Create an instance of InputData and create a memento. Confirm that the memento is not None and that it contains the initial stokes value in its DataList.
    - setMemento()
      - Create an instance of InputData. Set it’s form and add several variables. Create a memento for it and use it to call setMemento() on a new instance of InputData, InputDataNew. Create a memento of InputDataNew and confirm that the resulting dataList contains all of the added variables from the InputData memento.
  - For each of the data input states’s store() functions, a boolean value is returned. If the input is good, True is returned. If the input is bad, False is returned. Store this value in the variable success and use it to test good and bad input.
  - Reynolds
    - init()
      - Confirm the singleton instance reynolds is not none.

        - storeGoodVal()
            - Call store() with a valid reynolds number and navier stokes data. Assert success is True and that nStokesInputData contains the stored reynolds number.
        - storeBadVal()
            - Call store() with a string and assert success is False.
        - hasNext()
            - Assert hasNext() is True
        - next()
            - Assert next() is equal to State.Instance()
    - State
        - init()
            - Confirm the singleton instance state is not none.
        - storeGoodVall()
            - Call store() with steadyState and navier stokes data. Assert success is True and that nStokesInputData contains the stored state.
            - Call store() with transient and stokes data. Assert success is True and that stokesInputData contains the stored state.
            - Call store() with steadyState and stokes data. Assert success is True and that stokesInputData contains the stored state.
        - storeBadVal()
            - Call store() with an int and assert success is False.
            - Call store() with transient and navier stokes data. Assert success is False.
        - hasNext()
            - Assert hasNext() is True
        - next()
            - Assert next() is equal to MeshDimensions.Instance()
        - undo()
            - Assert undo() is equal to Reynolds.Instance()
    - MeshDimensions
        - init()
            - Confirm the singleton instance meshDims is not none.
        - storeGoodVall()
            - Call store() with valid mesh dimensions and navier stokes data. Assert success is True and that nStokesInputData contains the stored state.
        - storeBadVal()
            - Call store() with a string of wrong content and assert success is False.
            - Call store() with an int and assert success is False.
        - hasNext()
            - Assert hasNext() is True
        - next()
            - Assert next() is equal to Elements.Instance()

        - undo()
            - Assert undo() is equal to State.Instance()
    - Elements
        - init()
            - Confirm the singleton instance elements is not none.
        - storeGoodVall()
            - Call store() with a valid number of elements and navier stokes data. Assert success is True and that nStokesInputData contains the stored state.
        - storeBadVal()
            - Call store() with a string of wrong content and assert success is False.
            - Call store() with an int and assert success is False.
        - hasNext()
            - Assert hasNext() is True
        - next()
            - Assert next() is equal to PolyOrder.Instance()
        - undo()
            - Assert undo() is equal to MeshDimensions.Instance()
    - PolyOrder
        - init()
            - Confirm the singleton instance polyOrder is not none.
        - storeGoodVall()
            - Call store() with a valid polynomial order and navier stokes data. Assert success is True and that nStokesInputData contains the stored state.
        - storeBadVal()
            - Call store() with a string and assert success is False.
            - Call store() with a number greater than 9 and assert success is False.
            - Call store() with a double and assert success is False.
        - hasNext()
            - Assert hasNext() is True
        - next()
            - Assert next() is equal to Inflow.Instance()
        - undo()
            - Assert undo() is equal to Elements.Instance()
    - Inflow
        - init()
            - Confirm the singleton instance inflow is not none.
        - storeBadVal()
            - Call store() with a string and assert success is False.
            - Call store() with a double single and assert success is False.
        - obtainData()
            - Tested through store()
        - hasNext()
            - Assert hasNext() is True

        - next()
            - Assert next() is equal to Outflow.Instance()
        - undo()
            - Assert undo() is equal to PolyOrder.Instance()
    - Outflow
        - init()
            - Confirm the singleton instance outflow is not none.
        - storeBadVal()
            - Call store() with a string and assert success is False.
            - Call store() with a single double and assert success is False.
        - obtainData()
            - Tested through store()
        - hasNext()
            - Assert hasNext() is True
        - next()
            - Assert next() is equal to Walls.Instance()
        - undo()
            - Assert undo() is equal to Inflow.Instance()
    - Walls
        - init()
            - Confirm the singleton instance walls is not none.
        - storeBadVal()
            - Call store() with a string and assert success is False.
            - Call store() with a single double and assert success is False.
        - obtainData()
            - Tested through store()
        - hasNext()
            - Assert hasNext() is False
        - undo()
            - Assert undo() is equal to Outflow.Instance()
    - getFunction()
        - Confirm returns "undo" when input string is "undo". Confirm returns False when input string format is incorrect. Confirm adds proper Function to the list by evaluating and testing if returned values match appropriately (stringToFunction tested elsewhere).
    - getFilter()
        - Confirm returns "undo" when input string is "undo". Confirm returns False when input string format is incorrect. Confirm adds proper SpatialFilter to the list by testing if values match appropriately (stringToFilter tested elsewhere).
    - stringToDims()
        - Confirm float values added for x and y match the string passed as a parameter. Confirm raises exception when input string format is incorrect.
    - stringToElements()

        - Confirm int values added for x and y match the string passed as a parameter. Confirm raises exception when input string format is incorrect or contains float values.
- ParseFunction
    - add()
        - Define a simple function with addition. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - subtract()
        - Define a simple function with subtraction. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - divide()
        - Define a simple function with division. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - multiply()
        - Define a simple function with multiplication. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - exponent()
        - Define a simple function with exponentiation. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - negative()
        - Define a simple function with negation. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - parenMultiplty()
        - Define a simple function with parenthetical multiplication, 3(x). Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - xAndY()
        - Define a simple function with x and y as parameters. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
    - noParens()
        - Define a simple function with no parenthesis. Calculate the answer using basic python math operations. Confirm that the

parsed and evaluated answer to the PyCamellia function equals the calculated answer.

- doubles()
  - Define a simple function with doubles and integers. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
- halfDoubles()
  - Define a simple function with poorly inputted doubles, 1. or .4. Calculate the answer using basic python math operations. Confirm that the parsed and evaluated answer to the PyCamellia function equals the calculated answer.
- ePowerOfTen()
  - Define a simple function with e used in the power of ten.

- Plotter
  - Test this manually because of the visual output. Each test case is created to test output on formulations after various constructions/refinements, making sure that the visual output is the same as what is expected.
  - TestPlotter
    - test_plotMesh()
      - Creates a Stokes formulation and plots the msh.
    - test_plotRefineMesh()
      - Creates a Stokes formulation, refines it with a manual h refine on cell 0, and plots the mesh.
    - test_plot_u1()
      - Creates a Stokes formulation and plots the u1.
    - test_plotpAutoRefine_u1()
      - Creates a Stokes formulation, performs a p auto refine and plots the u1. Note this should be the same as above as p does not affect u1.
    - test_plothAutoRefine_u1()
      - Creates a Stokes formulation, performs an h auto refine and plots the u1. The h manual refine should refine the top half of the mesh.
    - test_plotpManualRefine_u1()
      - Creates a Stokes formulation, performs a p manual refine and plots the u1. Note this should be the same as above as p does not affect u1.
    - test_plothManualRefine_u1()
      - Creates a Stokes formulation, performs an h manual refine and plots the u1. The h manual refine is set to refine cells 0 and 1 (the left side of this mesh).
    - test_plot_u2()
      - Creates a Stokes formulation and plots the u2.
    - test_plotpAutoRefine_u2()

            - Creates a Stokes formulation, performs a p auto refine and plots the u2. Note this should be the same as above as p does not affect u1.
        - test_plothAutoRefine_u2()
            - Creates a Stokes formulation, performs an h auto refine and plots the u2. The h manual refine should refine the top half of the mesh.
        - test_plotpManualRefine_u2()
            - Creates a Stokes formulation, performs a p manual refine and plots the u2. Note this should be the same as above as p does not affect u1.
        - test_plothManualRefine_u2()
            - Creates a Stokes formulation, performs an h manual refine and plots the u2. The h manual refine is set to refine cells 0 and 1 (the left side of this mesh).
    - TestPlotterError
        - test_plot_energyError()
            - Creates a Stokes formulation and plots the energyError.
        - test_plotpAutoRefine_energyError()
            - Creates a Stokes formulation, performs a p auto refine and plots the energyError. Note this should be the same as above as p does not affect u1.
        - test_plothAutoRefine_energyError()
            - Creates a Stokes formulation, performs an h auto refine and plots the energyError. The h manual refine should refine the top half of the mesh.
        - test_plotpManualRefine_energyError()
            - Creates a Stokes formulation, performs a p manual refine and plots the energyError. Note this should be the same as above as p does not affect u1.
        - test_plothManualRefine_energyError()
            - Creates a Stokes formulation, performs an h manualrefine and plots the energyError. The h manual refine is set to refine cells 0 and 1 (the left side of this mesh).
    - TestPlotterStream
        - test_plot_streamPhi()
            - Creates a Stokes formulation and plots the streamPhi.
        - test_plotpAutoRefine_streamPhi()
            - Creates a Stokes formulation, performs a p auto refine and plots the streamPhi. Note this should be the same as above as p does not affect u1.
        - test_plothAutoRefine_streamPhi()
            - Creates a Stokes formulation, performs an h auto refine and plots the streamPhi. The h manual refine should refine the top half of the mesh.
        - test_plotpManualRefine_streamPhi()

            - Creates a Stokes formulation, performs a p manual refine and plots the streamPhi. Note this should be the same as above as p does not affect u1.
        - test_plothManualRefine_streamPhi()
            - Creates a Stokes formulation, performs an h manualrefine and plots the streamPhi. The h manual refine is set to refine cells 0 and 1 (the left side of this mesh).
    - TestPlotterP
        - test_plot_p()
            - Creates a Stokes formulation and plots the p.
        - test_plotpAutoRefine_p()
            - Creates a Stokes formulation, performs a p auto refine and plots the p. Note that the p should be refined in certain cells now.
        - test_plothAutoRefine_p()
            - Creates a Stokes formulation, performs an h auto refine and plots the p. The h manual refine should refine the top half of the mesh.
        - test_plotpManualRefine_p()
            - Creates a Stokes formulation, performs a p manual refine and plots the p. Note that the p should be refined on specific cells.
        - test_plothManualRefine_p()
            - Creates a Stokes formulation, performs an h manualrefine and plots the p. The h manual refine is set to refine cells 0 and 1 (the left side of this mesh).
- Solver
    - Create instances of all the states the machine can be in. Confirm that the state changes appropriately when act() is passed valid commands such as undo or data input or invalid data.