Chapter 1

Example problem: Spin-up of a viscous fluid

This is our first axisymmetric Navier–Stokes example problem. We discuss the non-dimensionalisation of the equations and their implementation in `oomph-lib`, and demonstrate the solution of a spin-up problem.

1.1 The axisymmetric Navier–Stokes equations

In dimensional form the axisymmetric Navier–Stokes equations are given by the momentum equations (for the r^* , z^* and θ directions, respectively)

$$\rho\left[\frac{\partial u_r^*}{\partial t^*}+u_r^*\frac{\partial u_r^*}{\partial r^*}-\frac{{u_\theta^*}^2}{r^*}+u_z^*\frac{\partial u_r^*}{\partial z^*}\right]=B_r^*\left(r^*,z^*,t^*\right)+\rho G_r^*+\frac{\partial \tau_{rr}^*}{\partial r^*}+\frac{\tau_{rr}^*}{r^*}-\frac{\tau_{\theta\theta}^*}{r^*}+\frac{\partial \tau_{rz}^*}{\partial z^*},$$

$$\rho\left[\frac{\partial u_z^*}{\partial t^*}+u_r^*\frac{\partial u_z^*}{\partial r^*}+u_z^*\frac{\partial u_z^*}{\partial z^*}\right]=B_z^*\left(r^*,z^*,t^*\right)+\rho G_z^*+\frac{\partial \tau_{zr}^*}{\partial r^*}+\frac{\tau_{zr}^*}{r^*}+\frac{\partial \tau_{zz}^*}{\partial z^*},$$

$$\rho\left[\frac{\partial u_\theta^*}{\partial t^*}+u_r^*\frac{\partial u_\theta^*}{\partial r^*}+\frac{u_r^*u_\theta^*}{r^*}+u_z^*\frac{\partial u_\theta^*}{\partial z^*}\right]=B_\theta^*\left(r^*,z^*,t^*\right)+\rho G_\theta^*+\frac{\partial \tau_{\theta r}^*}{\partial r^*}+\frac{\tau_{\theta r}^*}{r^*}+\frac{\tau_{r\theta}^*}{r^*}+\frac{\partial \tau_{\theta z}^*}{\partial z^*},$$

and the continuity equation

$$\frac{\partial u_r^*}{\partial r^*}+\frac{u_r^*}{r^*}+\frac{\partial u_z^*}{\partial z^*}=Q^*,$$

where u_r^* , u_z^* and u_θ^* are the radial, axial and azimuthal velocity components respectively, p^* is the pressure and t^* is time. We have split the body force into two components: A constant vector $\rho\, G_i^*$ (where $i=r,z,\theta$) which typically represents gravitational forces; and a variable body force, $B_i^*(r^*,z^*,t^*)$. $Q^*(r^*,z^*,t^*)$ is a volumetric source term for the continuity equation and is typically equal to zero.
The components of the dimensional stress tensor τ_{ij}^* are defined as:

$$\tau_{rr}^*=-p^*+\mu(1+\Gamma)\frac{\partial u_r^*}{\partial r^*},\qquad \tau_{\theta\theta}^*=-p^*+\mu(1+\Gamma)\frac{u_r^*}{r^*},\qquad \tau_{zz}^*=-p^*+\mu(1+\Gamma)\frac{\partial u_z^*}{\partial z^*},$$

$$\tau_{rz}^*=\mu\left(\frac{\partial u_r^*}{\partial z^*}+\Gamma\frac{\partial u_z^*}{\partial r^*}\right),\qquad \tau_{zr}^*=\mu\left(\frac{\partial u_z^*}{\partial r^*}+\Gamma\frac{\partial u_r^*}{\partial z^*}\right),$$

$$\tau_{r\theta}^*=\mu\left(\Gamma\frac{\partial u_\theta^*}{\partial r^*}-\frac{u_\theta^*}{r^*}\right),\qquad \tau_{\theta r}^*=\mu\left(\frac{\partial u_\theta^*}{\partial r^*}-\Gamma\frac{u_\theta^*}{r^*}\right),$$

$$\tau_{\theta z}^*=\mu\frac{\partial u_\theta^*}{\partial z^*},\qquad \tau_{z\theta}^*=\mu\Gamma\frac{\partial u_\theta^*}{\partial z^*}.$$

We note that taking $\Gamma = 1$ corresponds to using the stress-divergence form of the viscous term in the Navier–Stokes equations, which is the form that `oomph-lib` uses by default. We can, however, recover the 'standard' form by setting $\Gamma = 0$.

We non-dimensionalise the equations, using problem-specific reference quantities for the velocity, $\mathcal{U}$, length, $\mathcal{L}$, and time, $\mathcal{T}$, and scale the constant body force vector on the gravitational acceleration, g, so that

$$u_r^* = \mathcal{U}\, u_r, \qquad u_z^* = \mathcal{U}\, u_z, \qquad u_\theta^* = \mathcal{U}\, u_\theta,$$
$$r^* = \mathcal{L}\, r, \qquad z^* = \mathcal{L}\, z, \qquad t^* = \mathcal{T}\, t, \qquad G_i^* = g\, G_i,$$
$$p^* = \frac{\mu_{ref}\mathcal{U}}{\mathcal{L}}\, p, \qquad B_i^* = \frac{\mathcal{U}\mu_{ref}}{\mathcal{L}^2}\, B_i, \qquad Q^* = \frac{\mathcal{U}}{\mathcal{L}}\, Q,$$

where we note that the pressure and the variable body force have been non-dimensionalised on the viscous scale. μ_{ref} and ρ_{ref} are reference values for the fluid viscosity and density, respectively. In single-fluid problems, they are identical to the viscosity μ and density ρ of the (one and only) fluid in the problem.

The non-dimensional form of the axisymmetric Navier–Stokes equations is then given by

$$R_\rho Re\left[St\frac{\partial u_r}{\partial t} + u_r\frac{\partial u_r}{\partial r} - \frac{u_\theta^2}{r} + u_z\frac{\partial u_r}{\partial z}\right] = B_r\,(r,z,t) + R_\rho\frac{Re}{Fr}G_r + \frac{\partial \tau_{rr}}{\partial r} + \frac{\tau_{rr}}{r} - \frac{\tau_{\theta\theta}}{r} + \frac{\partial \tau_{rz}}{\partial z},$$
$$R_\rho Re\left[St\frac{\partial u_z}{\partial t} + u_r\frac{\partial u_z}{\partial r} + u_z\frac{\partial u_z}{\partial z}\right] = B_z\,(r,z,t) + R_\rho\frac{Re}{Fr}G_z + \frac{\partial \tau_{zr}}{\partial r} + \frac{\tau_{zr}}{r} + \frac{\partial \tau_{zz}}{\partial z},$$
$$R_\rho Re\left[St\frac{\partial u_\theta}{\partial t} + u_r\frac{\partial u_\theta}{\partial r} + \frac{u_r u_\theta}{r} + u_z\frac{\partial u_\theta}{\partial z}\right] = B_\theta\,(r,z,t) + R_\rho\frac{Re}{Fr}G_\theta + \frac{\partial \tau_{\theta r}}{\partial r} + \frac{\tau_{\theta r}}{r} + \frac{\tau_{r\theta}}{r} + \frac{\partial \tau_{\theta z}}{\partial z},$$

and

$$\frac{\partial u_r}{\partial r} + \frac{u_r}{r} + \frac{\partial u_z}{\partial z} = Q.$$

Here the components of the non-dimensional stress tensor τ_{ij} are defined as:

$$\tau_{rr} = -p + R_\mu(1+\Gamma)\frac{\partial u_r}{\partial r}, \qquad \tau_{\theta\theta} = -p + R_\mu(1+\Gamma)\frac{u_r}{r}, \qquad \tau_{zz} = -p + R_\mu(1+\Gamma)\frac{\partial u_z}{\partial z},$$
$$\tau_{rz} = R_\mu\left(\frac{\partial u_r}{\partial z} + \Gamma\frac{\partial u_z}{\partial r}\right), \qquad \tau_{zr} = R_\mu\left(\frac{\partial u_z}{\partial r} + \Gamma\frac{\partial u_r}{\partial z}\right),$$
$$\tau_{r\theta} = R_\mu\left(\Gamma\frac{\partial u_\theta}{\partial r} - \frac{u_\theta}{r}\right), \qquad \tau_{\theta r} = R_\mu\left(\frac{\partial u_\theta}{\partial r} - \Gamma\frac{u_\theta}{r}\right),$$
$$\tau_{\theta z} = R_\mu\frac{\partial u_\theta}{\partial z} \qquad \tau_{z\theta} = R_\mu\Gamma\frac{\partial u_\theta}{\partial z}.$$

The dimensionless parameters

$$Re = \frac{\mathcal{U}\mathcal{L}\rho_{ref}}{\mu_{ref}}, \qquad St = \frac{\mathcal{L}}{\mathcal{U}\mathcal{T}}, \qquad Fr = \frac{\mathcal{U}^2}{g\mathcal{L}},$$

are the Reynolds number, Strouhal number and Froude number respectively. $R_\rho = \rho/\rho_{ref}$ and $R_\mu = \mu/\mu_{ref}$ represent the ratios of the fluid's density and its dynamic viscosity, relative to the density and viscosity values used to form the non-dimensional parameters (By default, $R_\rho = R_\mu = 1$; other values tend to be used in problems involving multiple fluids). We refer to `another tutorial` for a more detailed discussion of these non-dimensional parameters and their default values.

The above equations are typically augmented by Dirichlet boundary conditions for (some of) the velocity components. On boundaries where no velocity boundary conditions are applied, the flow satisfies the "traction free" natural boundary condition $t_i = 0$. For example, in the spin-up problem to be considered below, no condition is applied to the z-component of the velocity on the symmetry boundary, which means that the traction in this direction, t_z, is equal to zero.

If the velocity is prescribed along the entire domain boundary, the fluid pressure p is only determined up to an arbitrary constant. This indeterminacy may be overcome by prescribing the value of the pressure at a single point in the domain. See the exercises at the end of the (non-axisymmetric) `driven cavity example` for further discussion of this issue.

1.2 Implementation

`oomph-lib` provides two LBB-stable isoparametric axisymmetric Navier–Stokes elements that are based on the `QElement<2,3>` family of geometric finite elements. They are nine-node quadrilateral elements which only differ in the way in which the pressure is represented. In `AxisymmetricQCrouzeixRaviartElements` the pressure is represented by a discontinuous, piecewise bi-linear function. `AxisymmetricQTaylorHood↩Elements` represent the pressure by a globally-continuous, piecewise bi-linear interpolation between the pressure values that are stored at the elements' four corner nodes.
The implementation of these axisymmetric Navier–Stokes elements is very similar to their non-axisymmetric counterparts, discussed in detail in another tutorial. The radial and axial nodal positions are stored at the first and second nodal coordinates respectively, and can therefore be accessed by the member functions

- Radial coordinate (r): `Node::x(0)`
- Axial coordinate (z): `Node::x(1)`

By default the radial, axial and azimuthal components are stored as the first, second and third nodal values respectively, and can therefore be accessed by the member functions

- Radial component (u_r): `Node::value(0)`
- Axial component (u_z): `Node::value(1)`
- Azimuthal component (u_θ): `Node::value(2)`

1.3 The example problem

The solution of the axisymmetric Navier–Stokes equations will be illustrated using the example of a spin-up problem. We consider a sealed cylindrical container of radius R and height H, filled with a fluid of density ρ and dynamic viscosity μ. Both the fluid and the cylinder are initially at rest, and at time $t = 0$ the cylinder immediately begins to rotate about its vertical axis of symmetry with a constant angular velocity Ω. Initially, the bulk of the fluid remains stationary, with the exception of the regions next to the solid boundaries. The fluid near the top and bottom 'lids' is moving faster than that along the bulk of the cylinder, and gets driven radially outward. It is replaced by fluid from the interior, setting up secondary flows in the r- z plane, until eventually the entire fluid is moving in solid body rotation with the cylinder.

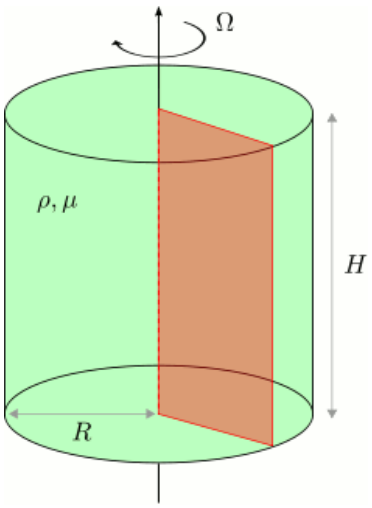

Figure 1.1 Sketch of the problem. The red `slice' represents the domain in which the axisymmetric Navier--Stokes equations will be solved.

We model this problem by solving the axisymmetric Navier–Stokes equations in a rectangular domain of width R and height H. For our non-dimensionalisation we choose the length scale $\mathcal{L}$ to be the radius of the cylinder R and the velocity scale $\mathcal{U}$ to be the speed of the outer wall $R\Omega$. The time scale $\mathcal{T}$ is chosen to be $\mathcal{L}/\mathcal{U}$ so that the

Strouhal number is equal to one. We choose an aspect ratio of $H = 1.4R$ and therefore our domain D is defined to be

$$D = \{r \in [0.0, 1.0],\ z \in [0.0, 1.4]\}.$$

The governing equations are subject to the Dirichlet boundary conditions

$$u_r = 0, \qquad u_z = 0, \qquad u_\theta = r$$

on the bottom, right and top boundaries and

$$u_r = 0, \qquad u_\theta = 0$$

on the left boundary, $r = 0$. The z-component of the velocity on this boundary is traction-free, which corresponds to the symmetry condition $\partial u_z/\partial r = 0$.

1.4 Results

The figure below shows contour plots of the azimuthal velocity component and the pressure distribution with superimposed streamlines, taken from an animation of the flow field, computed with axisymmetric Taylor-Hood elements for the parameters $Re = Re\,St = 5.0$

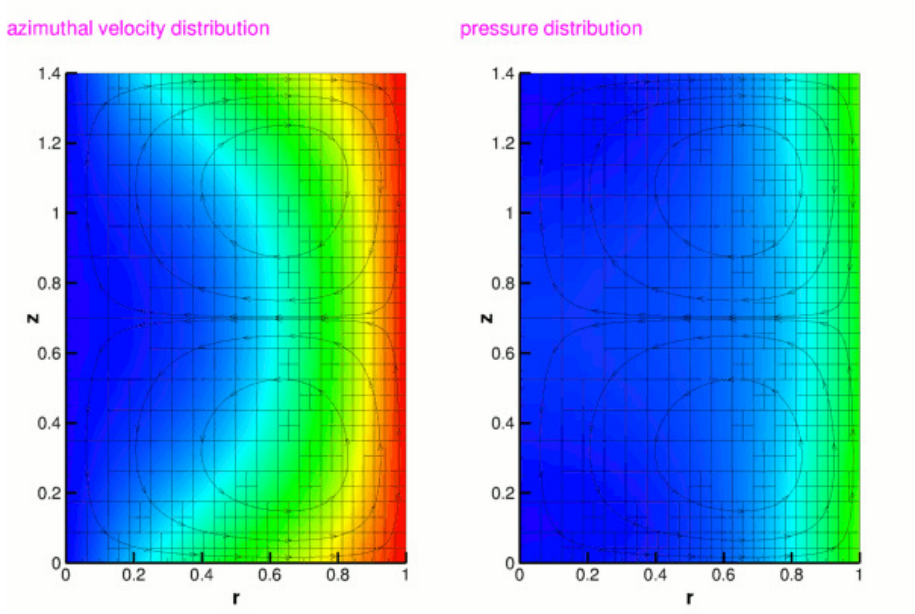

Figure 1.2 Plot of the azimuthal velocity and pressure fields.

The figure below shows carpet plots of all three velocity components and the pressure, taken from another animation of the flow field, computed with axisymmetric Taylor-Hood elements for the parameters $Re = Re\,St = 5.0$

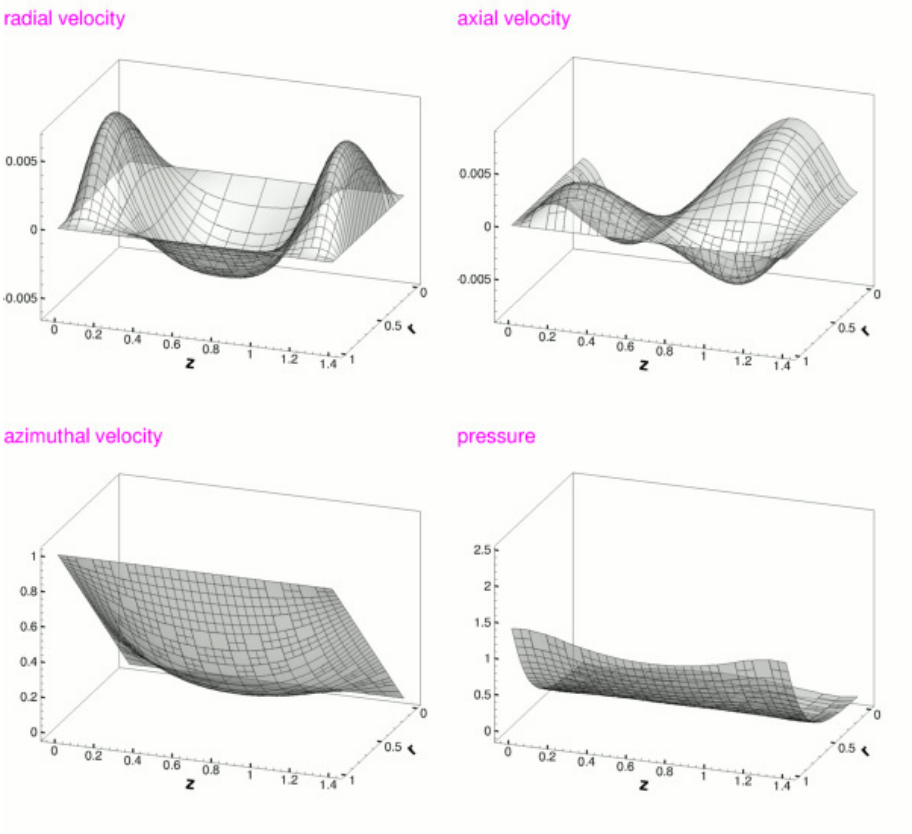

Figure 1.3 Plot of the velocity and pressure fields.

1.5 Global parameters and functions

The Reynolds number and the Womersley number (the product of the Reynolds and Strouhal numbers) are needed in this problem. As usual, we define them in a namespace:

```
//==start_of_namespace==============================================
/// Namespace for physical parameters
//==================================================================
namespace Global_Physical_Variables
{

 /// Reynolds number
 double Re = 5.0;

 /// Womersley number
 double ReSt = 5.0;
} // End of namespace
```

1.6 The driver code

We start by specifying the (non-dimensional) length of time we want to run the simulation for and the size of the timestep. Because all driver codes are run as part of `oomph-lib's` self-testing routines we allow the user to pass a command line argument to the executable which sets the maximum time to some lower value.

```
//==start_of_main==================================================
/// Driver code for axisymmetric spin-up problem
//==================================================================
int main(int argc, char* argv[])
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);

 /// Maximum time
 double t_max = 1.0;

 /// Duration of timestep
 const double dt = 0.01;
 // If we are doing validation run, use smaller number of timesteps
```

```
 if(CommandLineArgs::Argc>1) { t_max = 0.02; }
```

Next we specify the dimensions of the mesh and the number of elements in the radial and azimuthal directions.

```
 // Number of elements in radial (r) direction
 const unsigned n_r = 2;
 // Number of elements in axial (z) direction
 const unsigned n_z = 2;
 // Length in radial (r) direction
 const double l_r = 1.0;
 // Length in axial (z) direction
 const double l_z = 1.4;
```

We build the problem using `RefineableAxisymmetricQTaylorHoodElements` and the `BDF<2>` timestepper, before calling `unsteady_run(...)`. This function solves the system at each timestep using the `Problem::unsteady_newton_solve(...)` function before documenting the result.

```
 // RefineableAxisymmetricQTaylorHoodElements
 // -----------------------------------------
 {
  cout << "Doing RefineableAxisymmetricQTaylorHoodElement" << std::endl;
  // Build the problem with RefineableAxisymmetricQTaylorHoodElements
  RotatingCylinderProblem
   <RefineableAxisymmetricQTaylorHoodElement, BDF<2> >
   problem(n_r,n_z,l_r,l_z);

  // Solve the problem and output the solution
  problem.unsteady_run(t_max,dt,"RESLT_TH");
 }
```

We then repeat the process with `RefineableAxisymmetricQCrouzeixRaviartElements`.

```
 // RefineableAxisymmetricQCrouzeixRaviartElements
 // ----------------------------------------------
 {
  cout << "Doing RefineableAxisymmetricQCrouzeixRaviartElement" << std::endl;
  // Build the problem with RefineableAxisymmetricQCrouzeixRaviartElements
  RotatingCylinderProblem
   <RefineableAxisymmetricQCrouzeixRaviartElement, BDF<2> >
   problem(n_r,n_z,l_r,l_z);
  // Solve the problem and output the solution
  problem.unsteady_run(t_max,dt,"RESLT_CR");
 }

} // End of main
```

1.7 The problem class

The `Problem` class for our unsteady axisymmetric Navier–Stokes problem is very similar to that used in the `Rayleigh channel example`. We specify the type of the element and the type of the timestepper (assumed to be a member of the `BDF` family) as template parameters, and pass the number of elements and domain length in both coordinate directions to the problem constructor. We define an empty destructor, functions to set the initial and boundary conditions and a post-processing function `doc_solution(...)`, which will be used by the timestepping function `unsteady_run(...)`.

```
//==start_of_problem_class============================================
/// \short Refineable rotating cylinder problem in a rectangular
/// axisymmetric domain
//====================================================================
template <class ELEMENT, class TIMESTEPPER>
class RotatingCylinderProblem : public Problem
{
public:

 /// Constructor: Pass the number of elements and the lengths of the
 /// domain in the radial (r) and axial (z) directions
 RotatingCylinderProblem(const unsigned& n_r, const unsigned& n_z,
                         const double& l_r, const double& l_z);

 /// Destructor (empty)
 ~RotatingCylinderProblem() {}

 /// Set initial conditions
 void set_initial_condition();

 /// Set boundary conditions
 void set_boundary_conditions();

 /// Document the solution
 void doc_solution(DocInfo &doc_info);

 /// Do unsteady run up to maximum time t_max with given timestep dt
 void unsteady_run(const double& t_max, const double& dt,
                   const string dir_name);
```

Next we define an access function to the specific `Mesh`:

```
 /// Access function for the specific mesh
```

```
 RefineableRectangularQuadMesh<ELEMENT>* mesh_pt()
  {
   return dynamic_cast<RefineableRectangularQuadMesh<ELEMENT>*>
    (Problem::mesh_pt());
  }
```

We reset the boundary conditions before each solve by overloading `Problem::actions_before_newton_solve()`. This is to ensure that all newly-created nodes are given the correct boundary conditions.

```
private:

 /// \short Update the problem specs before solve.
 /// Reset velocity boundary conditions just to be on the safe side...
 void actions_before_newton_solve() { set_boundary_conditions(); }

 /// No actions required after solve step
 void actions_after_newton_solve() {}
```

In Navier–Stokes problems in which the velocity is prescribed along the entire domain boundary, the pressure is only determined up an arbitrary constant, making it necessary to "pin" one pressure value. If the pinned pressure degree of freedom is associated with an element that is unrefined during the mesh adaptation, the pinned degree of freedom may no longer exist in the adapted problem. We therefore use the function `Problem::actions_after_adapt()` to ensure that precisely one pressure degree of freedom is pinned when re-solving the adapted problem. Additionally, the possible presence of hanging nodes in an adapted mesh requires special treatment for elements (e.g. Taylor-Hood elements) in which the pressure is represented by a low-order interpolation between a subset of the element's nodal values. The function `AxisymmetricNavierStokesEquations::pin_redundant_nodal_pressures(...)` performs the required tasks. The technical details of these functions are discussed in detail in an earlier tutorial.

```
 /// \short After adaptation: Pin pressure again (the previously pinned
 /// value might have disappeared) and pin redudant pressure dofs
 void actions_after_adapt()
  {
   // Unpin all pressure dofs
   RefineableAxisymmetricNavierStokesEquations::
    unpin_all_pressure_dofs(mesh_pt()->element_pt());

   // Pin redudant pressure dofs
   RefineableAxisymmetricNavierStokesEquations::
    pin_redundant_nodal_pressures(mesh_pt()->element_pt());

   // Now set the pressure in first element at 'node' 0 to 0.0
   fix_pressure(0,0,0.0);
  } // End of actions_after_adapt
```

Finally, we provide a helper function `fix_pressure(...)` which pins a pressure value in a specified element and assigns a specific value.

```
 /// Fix pressure in element e at pressure dof pdof and set to pvalue
 void fix_pressure(const unsigned& e,
                   const unsigned& pdof,
                   const double& pvalue)
  {
   // Cast to actual element and fix pressure
   dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e))->
    fix_pressure(pdof,pvalue);
  }
}; // End of problem class
```

1.8 The problem constructor

We start by building the timestepper, determining its type from the class's second template argument, and pass a pointer to it to the problem, using the function `Problem::add_time_stepper_pt(...)`.

```
//==start_of_constructor=============================================
/// Constructor for refineable rotating cylinder problem
//====================================================================
template <class ELEMENT, class TIMESTEPPER>
RotatingCylinderProblem<ELEMENT,TIMESTEPPER>::
RotatingCylinderProblem(const unsigned& n_r, const unsigned& n_z,
                        const double& l_r, const double& l_z)
{
 // Allocate the timestepper (this constructs the time object as well)
 add_time_stepper_pt(new TIMESTEPPER);
```

Next we build the adaptive mesh and specify an error estimator, which will be used to guide the automatic mesh adaptation. We pass this to the mesh, set the maximum refinement level and override the maximum and minimum permitted errors, which are used to determine whether or not an element should be refined/unrefined during mesh adaptation.

```
 // Build and assign mesh
 Problem::mesh_pt() = new RefineableRectangularQuadMesh<ELEMENT>
  (n_r,n_z,l_r,l_z,time_stepper_pt());
 // Create and set the error estimator for spatial adaptivity
```

```
 mesh_pt()->spatial_error_estimator_pt() = new Z2ErrorEstimator;

 // Set the maximum refinement level for the mesh to 4
 mesh_pt()->max_refinement_level() = 4;
 // Override the maximum and minimum permitted errors
 mesh_pt()->max_permitted_error() = 1.0e-2;
 mesh_pt()->min_permitted_error() = 1.0e-3;
```

We pin the radial and azimuthal velocity components on all boundaries, and the axial component on the three solid boundaries.

```
 // Set the boundary conditions for this problem
 // ------------------------------------------
 // All nodes are free by default -- just pin the ones that have
 // Dirichlet conditions here
 // Determine number of mesh boundaries
 const unsigned n_boundary = mesh_pt()->nboundary();
 // Loop over mesh boundaries
 for(unsigned b=0;b<n_boundary;b++)
  {
   // Determine number of nodes on boundary b
   const unsigned n_node = mesh_pt()->nboundary_node(b);
   // Loop over nodes on boundary b
   for(unsigned n=0;n<n_node;n++)
    {
     // Pin values for radial velocity on all boundaries
     mesh_pt()->boundary_node_pt(b,n)->pin(0);
     // Pin values for axial velocity on all SOLID boundaries (b = 0,1,2)
     if(b!=3) { mesh_pt()->boundary_node_pt(b,n)->pin(1); }
     // Pin values for azimuthal velocity on all boundaries
     mesh_pt()->boundary_node_pt(b,n)->pin(2);
    } // End of loop over nodes on boundary b
  } // End of loop over mesh boundaries
```

We pass the pointers to the Reynolds and Womersley numbers, Re and $Re\,St$, and the pointer to the global time object (created when we called `Problem::add_time_stepper_pt(...)` above) to the elements. Because we know that the mesh will remain stationary we can disable the ALE formulation of the unsteady equations by calling `AxisymmetricNavierStokesEquations::disable_ALE()`. This suppresses the additional computation required to calculate the correction to the Eulerian time-derivative $\partial u/\partial t$ which is required if the mesh is moving, as discussed in detail in another tutorial.

```
 // Complete the problem setup to make the elements fully functional
 // ----------------------------------------------------------------
 // Determine number of elements in mesh
 const unsigned n_element = mesh_pt()->nelement();
 // Loop over the elements
 for(unsigned e=0;e<n_element;e++)
  {
   // Upcast from GeneralisedElement to the present element
   ELEMENT* el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e));
   // Set the Reynolds number
   el_pt->re_pt() = &Global_Physical_Variables::Re;
   // Set the Womersley number
   el_pt->re_st_pt() = &Global_Physical_Variables::ReSt;
   // The mesh remains fixed
   el_pt->disable_ALE();
  } // End of loop over elements
```

Since no traction boundary conditions are applied anywhere, the pressure is only determined up to an arbitrary constant. For the reasons discussed above we pin any redundant pressure degrees of freedom caused by hanging nodes and then ensure a unique solution by pinning a single pressure value. Finally, we set up the equation numbering scheme using the function `Problem::assign_eqn_numbers()`.

```
 // Pin redundant pressure dofs
 RefineableAxisymmetricNavierStokesEquations::
  pin_redundant_nodal_pressures(mesh_pt()->element_pt());

 // Now set the pressure in first element at 'node' 0 to 0.0
 fix_pressure(0,0,0.0);

 // Set up equation numbering scheme
 cout << "Number of equations: " << assign_eqn_numbers() << std::endl;

} // End of constructor
```

1.9 Initial conditions

The function `set_initial_condition()` sets the initial conditions for the problem by looping over all the nodes in the mesh and setting all velocity components to zero. No initial conditions are required for the pressure. We then call the function `Problem::assign_initial_values_impulsive()` which copies the current values at each of the nodes into the required number of history values for the timestepper in question. This corresponds to an impulsive start, as for all time $t \leq 0$ the fluid is at rest. At the first timestep, the solid domain

boundaries are immediately moving with a speed corresponding to their radial distance from the symmetry boundary. `Problem::set_initial_condition()` is called after each mesh adaptation on the first timestep only. This means that any newly-created nodes obtain their values from the actual (analytical) initial conditions rather than from interpolation of the values of previously-existing nodes.

```
//==start_of_set_initial_condition=======================================
/// \short Set initial conditions: Set all nodal velocities to zero and
/// initialise the previous velocities to correspond to an impulsive start
//========================================================================
template <class ELEMENT, class TIMESTEPPER>
void RotatingCylinderProblem<ELEMENT,TIMESTEPPER>::set_initial_condition()
{
 // Determine number of nodes in mesh
 const unsigned n_node = mesh_pt()->nnode();

 // Loop over all nodes in mesh
 for(unsigned n=0;n<n_node;n++)
  {
   // Loop over the three velocity components
   for(unsigned i=0;i<3;i++)
    {
     // Set velocity component i of node n to zero
     mesh_pt()->node_pt(n)->set_value(i,0.0);
    }
  }

 // Initialise the previous velocity values for timestepping
 // corresponding to an impulsive start
 assign_initial_values_impulsive();

} // End of set_initial_condition
```

1.10 Boundary conditions

The function `set_boundary_conditions()` sets the boundary conditions for the problem. On the three solid boundaries ($r = 1.0$, $z = 0.0$ and $z = 1.4$) we set the radial and axial velocities to zero so that there is no penetration of the wall by the fluid or flow along it. To simulate the domain rotating around the axis $r = 0$ we set the azimuthal velocity at each node along these boundaries to be equal to the radial position of the node. On the symmetry boundary ($r = 0$) we set the radial and azimuthal velocities to zero but leave the axial component unconstrained. As discussed above, not applying a velocity boundary condition causes the flow to satisfy the "traction free" natural boundary condition; in this case, $t_z = 0$. This corresponds to the symmetry condition $\partial u_z/\partial r = 0$.

```
//==start_of_set_boundary_conditions=====================================
/// \short Set boundary conditions: Set both velocity components to zero
/// on the bottom (solid) wall and the horizontal component only to zero
/// on the side (periodic) boundaries
//========================================================================
template <class ELEMENT, class TIMESTEPPER>
void RotatingCylinderProblem<ELEMENT,TIMESTEPPER>::set_boundary_conditions()
{
 // Determine number of mesh boundaries
 const unsigned n_boundary = mesh_pt()->nboundary();

 // Loop over mesh boundaries
 for(unsigned b=0;b<n_boundary;b++)
  {
   // Determine number of nodes on boundary b
   const unsigned n_node = mesh_pt()->nboundary_node(b);

   // Loop over nodes on boundary b
   for(unsigned n=0;n<n_node;n++)
    {
     // For the solid boundaries (boundaries 0,1,2)
     if(b<3)
      {
       // Get the radial component of position
       const double r_pos = mesh_pt()->boundary_node_pt(b,n)->x(0);

       // Set all velocity components to no flow along boundary
       mesh_pt()->boundary_node_pt(b,n)->set_value(0,0,0.0); // Radial
       mesh_pt()->boundary_node_pt(b,n)->set_value(0,1,0.0); // Axial
       mesh_pt()->boundary_node_pt(b,n)->set_value(0,2,r_pos); // Azimuthal
      }
     // For the symmetry boundary (boundary 3)
     if(b==3)
      {
       // Set only the radial (i=0) and azimuthal (i=2) velocity components
       // to no flow along boundary (axial component is unconstrained)
       mesh_pt()->boundary_node_pt(b,n)->set_value(0,0,0.0);
       mesh_pt()->boundary_node_pt(b,n)->set_value(0,2,0.0);
      }
```

```
    } // End of loop over nodes on boundary b
  } // End of loop over mesh boundaries

} // End of set_boundary_conditions
```

1.11 Post-processing

As expected, this member function documents the computed solution. We first print the value of the current time to the screen, before outputting the computed solution.

```
//==start_of_doc_solution=============================================
/// Document the solution
//====================================================================
template <class ELEMENT, class TIMESTEPPER>
void RotatingCylinderProblem<ELEMENT,TIMESTEPPER>::
doc_solution(DocInfo& doc_info)
{
 // Output the time
 cout << "Time is now " << time_pt()->time() << std::endl;
 ofstream some_file;
 char filename[100];
 // Set number of plot points (in each coordinate direction)
 const unsigned npts = 5;
 // Open solution output file
 sprintf(filename,"%s/soln%i.dat",
         doc_info.directory().c_str(),doc_info.number());
 some_file.open(filename);
 // Output solution to file
 mesh_pt()->output(some_file,npts);
 // Close solution output file
 some_file.close();

} // End of doc_solution
```

1.12 The timestepping loop

The function `unsteady_run(...)` is used to perform the timestepping procedure. We start by creating a `DocInfo` object to store the output directory and the label for the output files.

Before using any of `oomph-lib's` timestepping functions, the timestep dt must be passed to the problem's timestepping routines by calling the function `Problem::initialise_dt(...)` which sets the weights for all timesteppers in the problem. Next we assign the initial conditions by calling `Problem::set_initial_condition()`, which was discussed above.

```
 // Initialise timestep
 initialise_dt(dt);
 // Set initial condition
 set_initial_condition();
```

We define the maximum number of spatial adaptations which are permitted per timestep, and refine the mesh uniformly twice.

```
 // Maximum number of spatial adaptations per timestep
 unsigned max_adapt = 4;
 // Call refine_uniformly twice
 for(unsigned i=0;i<2;i++) { refine_uniformly(); }
```

We determine the number of timesteps to be performed and document the initial conditions. A flag, `first_timestep`, is initialised and set to true. This flag will be passed to `Problem::unsteady_newton_solve(...)`, and when set to true instructs the code to re-assign the initial conditions after every mesh adaptation.

```
 // Determine number of timesteps
 const unsigned n_timestep = unsigned(t_max/dt);
 // Doc initial solution
 doc_solution(doc_info);
 // Increment counter for solutions
 doc_info.number()++;
 // Are we on the first timestep? At this point, yes!
 bool first_timestep = true;
```

A key feature of this problem is the fact that the flow field approaches a "trivial" solution (rigid body rotation) which can be fully-resolved by the discretisation. In that case, equidistribution of the error (normalised by the norm of the global error which tends to zero!) leads to strong uniform mesh refinement despite the fact that the solution is fully converged. To avoid this, we prescribe a constant reference flux to normalise the error. For more details, see the discussion in the Z2ErrorEstimator class reference.

```
 // Specify normalising factor explicitly
 Z2ErrorEstimator* error_pt = dynamic_cast<Z2ErrorEstimator*>
  (mesh_pt()->spatial_error_estimator_pt());
 error_pt->reference_flux_norm() = 0.01;
```

Finally, we perform the actual timestepping loop. For each timestep the function `unsteady_newton_`

`solve(dt)` is called and the solution documented.

```
 // Timestepping loop
 for(unsigned t=1;t<=n_timestep;t++)
  {
   // Output current timestep to screen
   cout « "\nTimestep " « t « " of " « n_timestep « std::endl;
   // Take fixed timestep with spatial adaptivity
   unsteady_newton_solve(dt,max_adapt,first_timestep);

   // No longer on first timestep, so set first_timestep flag to false
   first_timestep = false;
   // Reset maximum number of adaptations for all future timesteps
   max_adapt = 1;

   // Doc solution
   doc_solution(doc_info);

   // Increment counter for solutions
   doc_info.number()++;

  } // End of timestepping loop

} // End of unsteady_run
```

1.13 Comments and Exercises

1.13.1 Good practice: Assigning boundary conditions

In our driver code we reset the boundary conditions for the problem before each Newton solve. This is done to ensure that any new boundary nodes created during mesh refinement are explicitly given the correct boundary conditions. However, the function that actually creates the new nodes, `RefineableQElement<2>::build(...)`, automatically assigns new nodes with values by interpolating within the father element. Since in our case the boundary conditions are linear, there is in fact no need to reset them at any point during the simulation, as the new boundary nodes were already given precisely the correct values by interpolation. Resetting the boundary conditions is only strictly necessary, therefore, in cases where:

1. The boundary conditions are given by a function which is of higher order than the shape functions used by the finite element, or
2. The boundary conditions are time-dependent.

1.13.2 Good practice: Assigning initial conditions

Similarly, we repeatedly call the `set_initial_condition()` function after each mesh adaptation during the first timestep. This is done to ensure that the exact initial conditions are given to newly created nodes during mesh refinement. Again, this is not strictly necessary in our case since `RefineableQElement<2>::build(...)` provides newly-created nodes with history values which are computed by interpolation of the history values stored at the nodes of the father element. Our initial conditions are constant and can therefore be represented exactly by this procedure. Should the initial conditions be given by an analytical function of higher order than the shape functions used by the finite elements, however, it would indeed be necessary to explicitly provide newly-created nodes with the exact initial conditions during the first timestep. For a more in-depth discussion, see another tutorial.

A slight subtlety

Omitting the re-assignment of the initial conditions on the adapted mesh when performing the first timestep does have a subtle effect that we encourage you to explore in the exercises below. You will observe that the solution obtained when re-assigning the initial conditions differs **very** slightly from that obtained when this step is suppressed. This obviously seems to contradict the statements made above and requires some explanation.
To understand why the behaviour is not unexpected (and perfectly acceptable!) let us analyse in more detail what really happens when we compute the first timestep in a spatially adaptive computation. When we perform the first Newton solve on the original mesh, the history values (which, for a BDF timestepper, represent the solution at previous timesteps) are identically equal to zero. The current values are also zero but this has no particular significance – they simply provide the initial guess for the solution at the advanced time level and their values are subsequently updated by the Newton solver.
Following the convergence of the Newton solver, the history values will therefore have retained their original values (zero) while the current values will have been updated to represent the "correct" solution of the nonlinear problem at the next timestep (on the current mesh). The accuracy of this solution is now assessed by the spatial error estimator.

If the estimated error is deemed too large, the mesh is adapted and all quantities (history values **and** current values) are automatically transferred onto the new mesh by interpolation – exactly as discussed above. Interpolation of the (identically equal to zero) history values onto the new mesh assigns zero history values for all newly-created nodes – exactly what we (needlessly) do in our own implementation of `Problem::set_initial_conditions()`. The interpolation of the current values transfers what is our current "best guess" for the solution at the advanced time onto the new mesh. In principle, this provides a "better" initial guess for the solution at the advanced time level than the zero initial guess that we re-assign when we call `Problem::set_initial_conditions()` but (as long as the Newton iteration converges) this assignment is irrelevant. However, starting the Newton iteration from a different initial guess will almost certainly lead to a slightly different solution – a solution being defined as a(ny) set of current values for which the maximum residual in the `Problem's` residual vector is less than the required tolerance. Hence, even though omitting or performing the re-assignment of the initial conditions leads to two **slightly** different solutions, both solutions are equally acceptable (within the threshold that is implicit in our convergence criterion for the Newton iteration).

1.13.3 Exercises

1. Remove the function `set_initial_condition()` entirely from the driver code (do not just leave it empty!), so that the initial conditions are not constantly reset during the first timestep. Confirm that the code still produces approximately (but not precisely) the same results, as discussed in the previous section.
2. Restore `set_initial_condition()` to its original state, but remove the call to `assign_initial_values_impulsive()`. Confirm that the code still produces precisely the same results. Why are exactly the same results produced this time?

1.14 Source files for this tutorial

- The source files for this tutorial are located in the directory:

  demo_drivers/axisym_navier_stokes/spin_up/

- The driver code is:

  demo_drivers/axisym_navier_stokes/spin_up/spin_up.cc

1.15 PDF file

A pdf version of this document is available.