Chapter 1

Demo problem: Relaxation oscillations of an interface between two viscous fluids in an axisymmetric domain

In an earlier example we considered a free surface Navier–Stokes problem in which the interface between two viscous fluids was deformed to a prescribed shape and then allowed to relax. This example considers a similar problem in which the two fluids occupy a cylindrical domain which is orientated such that its axis of symmetry is normal to the equilibrium position of the interface. The problem is assumed to be axisymmetric and so the computational domain is rectangular as before, with the difference here that the governing equations are the axisymmetric Navier–Stokes equations as opposed to the two-dimensional Cartesian Navier–Stokes equations employed by the other example. These equations are discussed in detail in another tutorial. Otherwise, the same overall strategy is employed. Since the implementation of the Navier–Stokes elements in `oomph-lib` is based on the Arbitrary Lagrangian Eulerian form of the equations, we can discretise the computational domain using a boundary-fitted mesh which deforms in response to the change in position of the interface. The positions of the nodes in the 'bulk' of the mesh are determined by treating the interior of the mesh as a fictitious elastic solid and solving a solid mechanics problem, a technique which we refer to as a 'pseudo-solid node update strategy'. The deformation of the free surface boundary itself is imposed by solving the kinematic boundary condition for a field of Lagrange multipliers at the interface, and this condition is discretised by attaching `FaceElements` to those 'bulk' elements in the 'lower' of the two fluids which have boundaries adjacent to the interface.

1.1 Choosing an appropriate interface deformation

The two-dimensional problem was started impulsively from a set of initial conditions such that the fluid was at rest and the interface was prescribed to be a cosine curve. Since this was an eigenmode of the system, the shape of the interface remained 'in mode' throughout the simulation. Choosing a single eigenmode as an initial condition allowed the results of the simulation to be compared with an analytical solution, and as we wish to do the same thing here we choose a zeroth-order Bessel function of the first kind, $J_0(kr)$, where k is a wavenumber and r is the radial spatial coordinate. This function has the property that its derivative with respect to r evaluated at $r = 0$ is zero, which must be the case if the interface is to be smooth at the symmetry boundary (the central axis of the cylinder).

A further consideration is that we want the equilibrium state of the system to be such that the interface is a flat line at $z = 0$, where z is the axial coordinate. This state can only be reached if we choose an initial deformation $z = h(r)$ which is volume conserving, and we therefore require

$$\int_{r=0}^{r=a} h(r) r dr = 0$$

to be satisfied, where a is the radius of the cylindrical container (or the width of the computational domain). Since in our case $h(r) = J_0(kr)$, and we will choose a computational domain of width $a = 1$, our initial condition must therefore satisfy

$$\int_0^1 r\, J_0(kr)\, dr = 0.$$

Using the properties of Bessel functions we find that this condition is met if $J_1(k) = 0$, and hence the values of k are constrained to be zeroes of $J_1(k)$.

Furthermore, the properties of Bessel functions are such that the derivative of $J_0(kr)$ with respect to r, evaluated at any point along the r-axis which corresponds to a zero of $J_1(kr)$, is itself zero. The velocity boundary condition in the axial direction is therefore the traction-free condition $\partial u_z/\partial r = 0$. This could be physically realised by having a 'slippery' outer wall where the contact line can move but the contact angle is fixed at 90^o, and is in fact an identical condition to the symmetry boundary condition prescribed at the axis of symmetry ($r = 0$).

1.2 The example problem

We will illustrate the solution of the unsteady axisymmetric Navier–Stokes equations using the example of a distorted interface between two viscous fluids which is allowed to relax. The domain is symmetric about the line $r = 0$, which corresponds to the axis of the cylindrical container.

The unsteady axisymmetric Navier–Stokes equations either side of a distorted interface.

Solve

$$Re\left[St\frac{\partial u_r}{\partial t}+u_r\frac{\partial u_r}{\partial r}-\frac{u_\theta^2}{r}+u_z\frac{\partial u_r}{\partial z}\right]=-\frac{\partial p}{\partial r}+\frac{Re}{Fr}G_r+\left[\frac{\partial^2 u_r}{\partial r^2}+\frac{1}{r}\frac{\partial u_r}{\partial r}-\frac{u_r}{r^2}+\frac{\partial^2 u_r}{\partial z^2}\right],$$

$$Re\left[St\frac{\partial u_\theta}{\partial t}+u_r\frac{\partial u_\theta}{\partial r}+\frac{u_r u_\theta}{r}+u_z\frac{\partial u_\theta}{\partial z}\right]=\frac{Re}{Fr}G_\theta+\left[\frac{\partial^2 u_\theta}{\partial r^2}+\frac{1}{r}\frac{\partial u_\theta}{\partial r}-\frac{u_\theta}{r^2}+\frac{\partial^2 u_\theta}{\partial z^2}\right],$$

$$Re\left[St\frac{\partial u_z}{\partial t}+u_r\frac{\partial u_z}{\partial r}+u_z\frac{\partial u_z}{\partial z}\right]=-\frac{\partial p}{\partial z}+\frac{Re}{Fr}G_z+\left[\frac{\partial^2 u_z}{\partial r^2}+\frac{1}{r}\frac{\partial u_z}{\partial r}+\frac{\partial^2 u_z}{\partial z^2}\right],$$

and

$$\frac{\partial u_r}{\partial r}+\frac{u_r}{r}+\frac{\partial u_z}{\partial z}=0 \qquad (1)$$

in the 'lower' fluid, and

$$R_\rho Re\left[St\frac{\partial u_r}{\partial t}+u_r\frac{\partial u_r}{\partial r}-\frac{u_\theta^2}{r}+u_z\frac{\partial u_r}{\partial z}\right]=-\frac{\partial p}{\partial r}+R_\rho\frac{Re}{Fr}G_r+R_\mu\left[\frac{\partial^2 u_r}{\partial r^2}+\frac{1}{r}\frac{\partial u_r}{\partial r}-\frac{u_r}{r^2}+\frac{\partial^2 u_r}{\partial z^2}\right],$$

$$R_\rho Re\left[St\frac{\partial u_\theta}{\partial t}+u_r\frac{\partial u_\theta}{\partial r}+\frac{u_r u_\theta}{r}+u_z\frac{\partial u_\theta}{\partial z}\right]=R_\rho\frac{Re}{Fr}G_\theta+R_\mu\left[\frac{\partial^2 u_\theta}{\partial r^2}+\frac{1}{r}\frac{\partial u_\theta}{\partial r}-\frac{u_\theta}{r^2}+\frac{\partial^2 u_\theta}{\partial z^2}\right],$$

$$R_\rho Re\left[St\frac{\partial u_z}{\partial t}+u_r\frac{\partial u_z}{\partial r}+u_z\frac{\partial u_z}{\partial z}\right]=-\frac{\partial p}{\partial z}+R_\rho\frac{Re}{Fr}G_z+R_\mu\left[\frac{\partial^2 u_z}{\partial r^2}+\frac{1}{r}\frac{\partial u_z}{\partial r}+\frac{\partial^2 u_z}{\partial z^2}\right],$$

and

$$\frac{\partial u_r}{\partial r}+\frac{u_r}{r}+\frac{\partial u_z}{\partial z}=0 \qquad (2)$$

in the 'upper' fluid. Gravity acts in the negative z direction and so $G_r = G_\theta = 0$ and $G_z = 1$. The governing equations are subject to the no slip boundary conditions

$$u_r = u_\theta = u_z = 0 \qquad (3)$$

on the top ($z = 2.0$) and bottom ($z = 0.0$) solid boundaries and the symmetry boundary conditions

$$u_r = u_\theta = 0 \qquad (4)$$

on the left ($r = 0.0$) and right ($r = 1.0$) boundaries.
We denote the position vector to the interface between the two fluids by $\mathbf{R}$, which is subject to the kinematic condition

$$\left(u_i - St\,\frac{\partial R_i}{\partial t}\right)n_i = 0, \qquad (5)$$

and the dynamic condition

$$\tau_{ij}^{[2]}n_j = \tau_{ij}^{[1]}n_j + \frac{1}{Ca}\kappa n_i. \qquad (6)$$

The non-dimensional, symmetric stress tensor in the 'lower' fluid is defined as

$$\tau_{rr} = -p + 2\frac{\partial u_r}{\partial r}, \qquad \tau_{\theta\theta} = -p + 2\frac{u_r}{r},$$
$$\tau_{zz} = -p + 2\frac{\partial u_z}{\partial z}, \qquad \tau_{rz} = \frac{\partial u_r}{\partial z} + \frac{\partial u_z}{\partial r}, \tag{7}$$
$$\tau_{\theta r} = r\frac{\partial}{\partial r}\left(\frac{u_\theta}{r}\right), \qquad \tau_{\theta z} = \frac{\partial u_\theta}{\partial z},$$

and that in the 'upper' fluid is defined as

$$\tau_{rr} = -p + 2R_\mu\frac{\partial u_r}{\partial r}, \qquad \tau_{\theta\theta} = -p + 2R_\mu\frac{u_r}{r},$$
$$\tau_{zz} = -p + 2R_\mu\frac{\partial u_z}{\partial z}, \qquad \tau_{rz} = R_\mu\left(\frac{\partial u_r}{\partial z} + \frac{\partial u_z}{\partial r}\right), \tag{8}$$
$$\tau_{\theta r} = R_\mu r\frac{\partial}{\partial r}\left(\frac{u_\theta}{r}\right), \qquad \tau_{\theta z} = R_\mu\frac{\partial u_\theta}{\partial z}.$$

The initial shape of the interface is defined by

$$\mathbf{R} = r\,\mathbf{e}_r + [1.0 + \epsilon J_0(kr)]\;\mathbf{e}_z, \tag{9}$$

where ϵ is the amplitude of the initial deflection, k is a wavenumber and $J_0(kr)$ is a zeroth-order Bessel function of the first kind.

1.3 Results

The figure below shows a contour plot of the pressure distribution taken from an animation of the flow field, for the parameters $Re = Re\,St = Re/Fr = 5.0$, $R_\rho = 0.5$, $R_\mu = 0.1$ and $Ca = 0.01$.

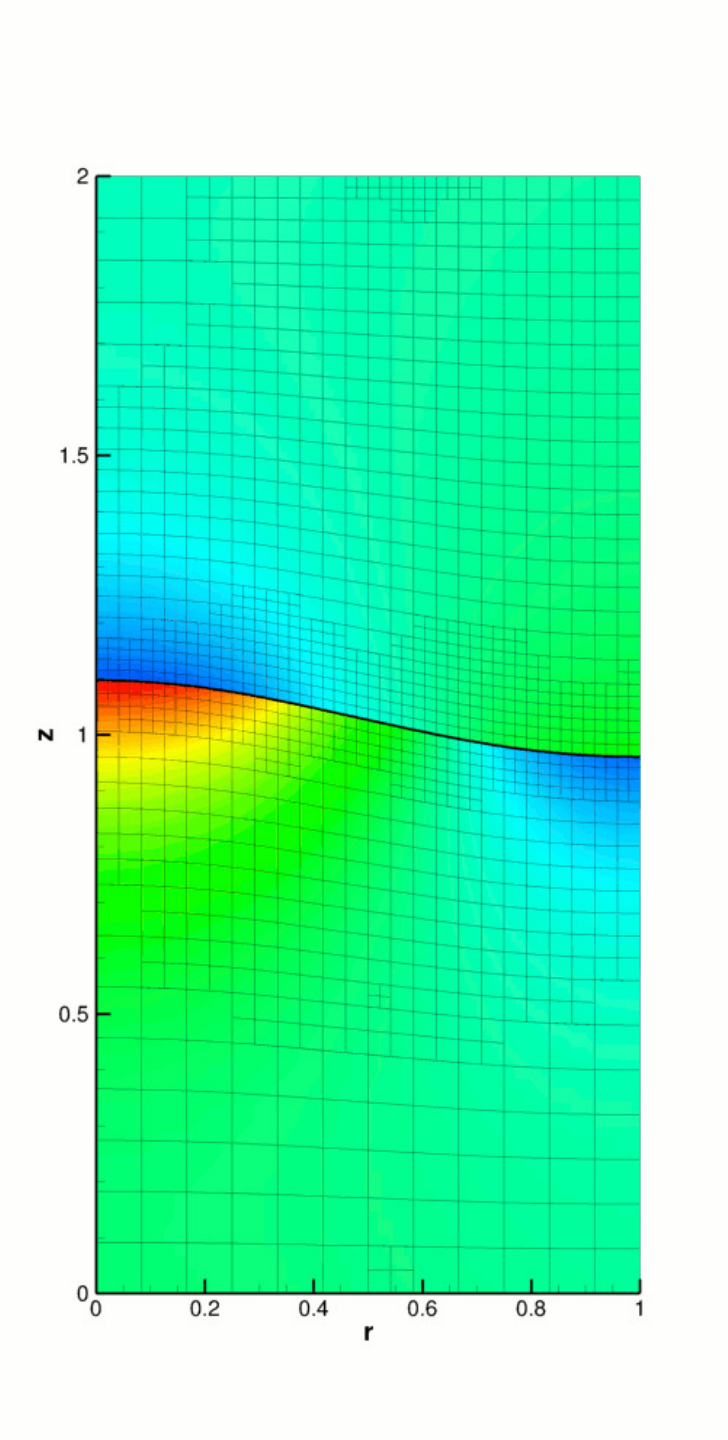

Figure 1.1 Pressure contour plot for the axisymmetric relaxing interface problem.

The restoring forces of surface tension and gravitational acceleration act to revert the interface to its undeformed flat state. The interface oscillates up and down, but the motion is damped as the energy in the system is dissipated through viscous forces. Eventually the interface settles down to its equilibrium position, as can be seen in the following time-trace of the height of the interface at the left-hand edge of the domain ($r = 0$).

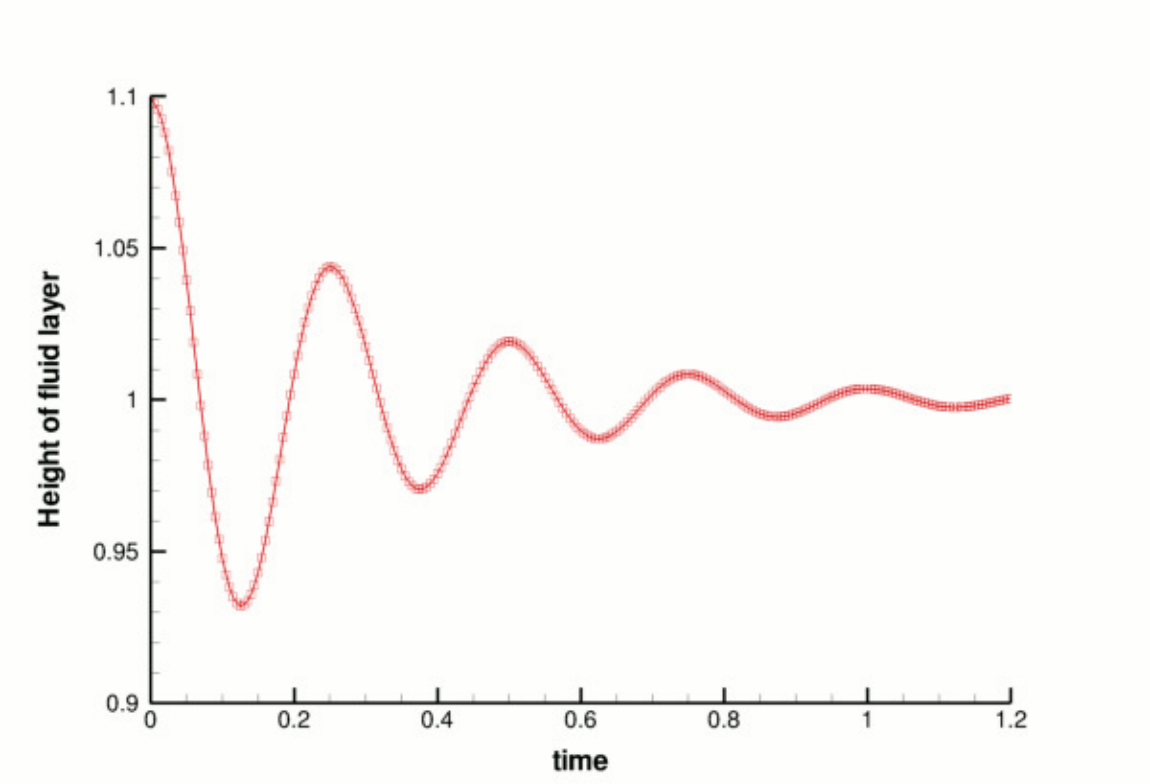

Figure 1.2 Time-trace of the height of the interface at the symmetry axis (r=0).

1.4 Validation

The free surface boundary conditions for the axisymmetric Navier–Stokes equations have been validated against an analytical test case, and we present the results in the figure below. For sufficiently small amplitudes, $\epsilon \ll 1$, we can linearise the governing equations and obtain a dispersion relation $\lambda(k)$. In the `two-dimensional single-layer example` we discussed the derivation of such a dispersion relation, and the technique used for this problem is very similar. The major difference is that the proposed separable solution for the linearised equations is of the form

$$\hat{h}(r,t) = He^{\lambda t}J_0(kr),$$
$$\hat{u}_r(r,z,t) = U_r(z)e^{\lambda t}J_1(kr),$$
$$\hat{u}_z(r,z,t) = U_z(z)e^{\lambda t}J_0(kr),$$

and

$$\hat{p}(r,z,t) = P(z)e^{\lambda t}J_0(kr).$$

As in the `two-dimensional two-layer example`, we will have a linear system containing nine unknowns, rather than the five unknowns that arise from a single-fluid problem. The real and imaginary parts of λ correspond to the growth rate and the frequency of the oscillating interface respectively, and can be compared to numerical results computed for given values of the wavenumber k. We choose an initial deflection amplitude of $\epsilon = 0.01$ and determine the growth rate and frequency of the oscillation from a time-trace of the left-hand edge of the interface.

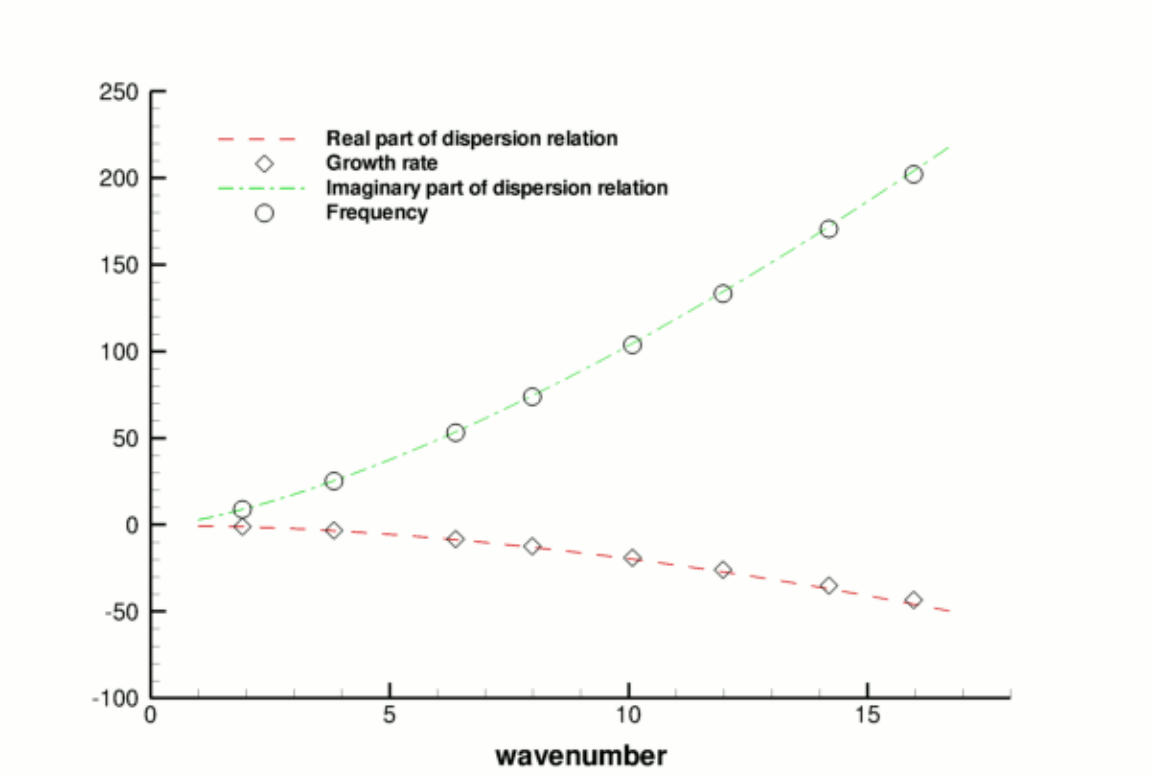

Figure 1.3 Validation of the code (points) by comparison with an analytical dispersion relation (lines).

1.5 Global parameters and functions

As in the earlier example, we use a namespace to define the dimensionless parameters Re, St, Re/Fr and Ca. Since we are solving for the unknown bulk nodal positions by treating the interior of the mesh as a fictitious elastic solid, we also need to define the Poisson ratio for this 'pseudo-solid's' generalised Hookean constitutive law, ν. As before, we define the density and viscosity ratios of the top fluid to the bottom fluid and a unit vector G which points in the direction of gravity. The only difference from before is that in this case G must have three dimensions since this is an axisymmetric problem.

```
 /// Direction of gravity
 Vector<double> G(3);
```

1.6 The driver code

The first section of the driver code is identical to that of the earlier example. We define a command line flag which allows us to run a 'validation' version of the code (for `oomph-lib's` self-testing routines) and check that the non-dimensional quantities provided in the `Global_Physical_Variables` namespace are self-consistent. Next we specify the duration of the simulation and the size of the timestep. If we are running the code as a validation, we set the length of the simulation such that only two timesteps are taken. The direction in which gravity acts is defined to be vertically downwards.

```
 // Set direction of gravity (vertically downwards)
 Global_Physical_Variables::G[0] = 0.0;
 Global_Physical_Variables::G[1] = -1.0;
 Global_Physical_Variables::G[2] = 0.0;
```

Finally, we build the problem using the 'pseudo-solid' version of `RefineableAxisymmetricQCrouzeix↩RaviartElements` 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.

```
 // Set up the elastic test problem with AxisymmetricQCrouzeixRaviartElements,
 // using the BDF<2> timestepper
 InterfaceProblem<RefineablePseudoSolidNodeUpdateElement<
 RefineableAxisymmetricQCrouzeixRaviartElement,
  RefineableQPVDElement<2,3> >,BDF<2> > problem;

 // Run the unsteady simulation
```

```
 problem.unsteady_run(t_max,dt);
} // End of main
```

1.7 The mesh class

The mesh class is almost identical to that in the `two-dimensional problem`. The only difference is that, given that this (specific) mesh is to be used in an axisymmetric problem, it does not make sense to allow the mesh to be periodic in the horizontal direction.

1.8 The problem class

The problem class is almost identical to the `two-dimensional problem`. The only modification arises in the arguments of `deform_free_surface(...)`, as the initial shape of the interface is a Bessel function $J_0(kr)$ as opposed to a cosine curve.

```
 /// Deform the mesh/free surface to a prescribed function
 void deform_free_surface(const double &epsilon, const double &k);
```

1.9 The problem constructor

The constructor starts by building the timestepper and setting the dimensions of the mesh. The number of elements in the r and z directions in both fluid layers are specified. The next section of the constructor is exactly as `before`. We build the bulk mesh, create an error estimator and set the maximum refinement level. An empty surface mesh is created and populated with a call to `create_interface_elements`, before the two meshes are combined to form the global mesh. Next we define the boundary conditions. On the top and bottom boundaries ($z = 2.0$ and $z = 0.0$) we apply the no-slip condition by pinning all three velocity components. On the outer (solid) wall ($r = 1.0$) we pin the radial (no penetration) and azimuthal (no slip) components but not the axial component. This simulates the "slippery" outer wall. These conditions are also applied at the symmetry boundary ($r = 0.0$). The vertical position of the nodes on the top and bottom boundaries are also pinned.

```
 // 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 = Bulk_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 = Bulk_mesh_pt->nboundary_node(b);
   // Loop over nodes on boundary b
   for(unsigned n=0;n<n_node;n++)
    {
     // Fluid boundary conditions:
     // -------------------------
     // Pin radial and azimuthal velocities (no slip/penetration)
     // on all boundaries other than the interface (b=4)
     if(b!=4)
      {
       Bulk_mesh_pt->boundary_node_pt(b,n)->pin(0);
       Bulk_mesh_pt->boundary_node_pt(b,n)->pin(2);
      }
     // Pin axial velocity on top (b=2) and bottom (b=0) boundaries
     // (no penetration). Because we have a slippery outer wall we do
     // NOT pin the axial velocity on this boundary (b=1); similarly,
     // we do not pin the axial velocity on the symmetry boundary (b=3).
     if(b==0 || b==2) { Bulk_mesh_pt->boundary_node_pt(b,n)->pin(1); }
     // Solid boundary conditions:
     // -------------------------
     // Pin vertical displacement on solid boundaries
     if(b==0 || b==2) { Bulk_mesh_pt->boundary_node_pt(b,n)->pin_position(1); }
    } // End of loop over nodes on boundary b
  } // End of loop over mesh boundaries
```

We pin the horizontal position of all nodes in the mesh as well as the azimuthal velocity components throughout the bulk of the domain (since in an axisymmetric problem these should remain zero always).

```
 // Loop over all nodes in mesh
 const unsigned n_node = Bulk_mesh_pt->nnode();
 for(unsigned n=0;n<n_node;n++)
  {
   // Pin horizontal displacement of all nodes
   Bulk_mesh_pt->node_pt(n)->pin_position(0);
   // Pin all azimuthal velocities throughout the bulk of the domain
   Bulk_mesh_pt->node_pt(n)->pin(2);
```

```
 }
```

The remainder of the problem constructor is identical to that of the two-dimensional problem. We create a generalised Hookean constitutive equation for the pseudo-solid mesh before looping over all elements in the bulk mesh and assigning pointers to the Reynolds and Womersley numbers, Re and $Re\,St$, the product of the Reynolds number and the inverse of the Froude number, Re/Fr, the direction of gravity, G, the constitutive law and the global time object. For those elements which correspond to the upper fluid layer, we also assign pointers for the viscosity and density ratios, R_μ and R_ρ. We pin one degree of freedom and call the function which assigns the boundary conditions, before finally setting up the equation numbering scheme.

1.10 Initial conditions

The `set_initial_conditions()` function is very similar to that in the two-dimensional problem. We loop over all nodes in the mesh and set all three velocity components to zero, before calling `Problem::assign_initial_values_impulsive()`. This function copies the current nodal values and positions into the required number of history values for the timestepper in question, simulating an impulsive start.

```
//==start_of_set_initial_condition=======================================
/// Set initial conditions: Set all nodal velocities to zero and
/// initialise the previous velocities and nodal positions to correspond
/// to an impulsive start
//========================================================================
template <class ELEMENT, class TIMESTEPPER>
void InterfaceProblem<ELEMENT,TIMESTEPPER>::set_initial_condition()
{
 // Determine number of nodes in mesh
 const unsigned n_node = Bulk_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
     Bulk_mesh_pt->node_pt(n)->set_value(i,0.0);
    }
  }

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

} // End of set_initial_condition
```

1.11 Boundary conditions

The `set_boundary_conditions()` function is very similar to that in the two-dimensional problem.

```
//==start_of_set_boundary_conditions=====================================
/// Set boundary conditions: Set all velocity components to zero
/// on the top and bottom (solid) walls and the radial and azimuthal
/// components only to zero on the side boundaries
//========================================================================
template <class ELEMENT, class TIMESTEPPER>
void InterfaceProblem<ELEMENT,TIMESTEPPER>::set_boundary_conditions()
{
 // Determine number of mesh boundaries
 const unsigned n_boundary = Bulk_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 = Bulk_mesh_pt->nboundary_node(b);

   // Loop over nodes on boundary b
   for(unsigned n=0;n<n_node;n++)
    {
     // Set radial component of the velocity to zero on all boundaries
     // other than the interface (b=4)
     if(b!=4) { Bulk_mesh_pt->boundary_node_pt(b,n)->set_value(0,0.0); }
     // Set azimuthal component of the velocity to zero on all boundaries
     // other than the interface (b=4)
     if(b!=4) { Bulk_mesh_pt->boundary_node_pt(b,n)->set_value(2,0.0); }
     // Set axial component of the velocity to zero on solid boundaries
     if(b==0 || b==2)
      {
       Bulk_mesh_pt->boundary_node_pt(b,n)->set_value(1,0.0);
      }
```

```
    }
  }
} // End of set_boundary_conditions
```

1.12 Actions before and after adaptation

These functions follow exactly the same structure as those in the two-dimensional problem.

1.13 Create interface elements

This function is identical to that in the two-dimensional problem, with the exception that instead of creating interface elements of the type
`ElasticLineFluidInterfaceElement<ELEMENT>`,
we are creating interface elements of the type
`ElasticAxisymFluidInterfaceElement<ELEMENT>`,
where `ELEMENT` is the templated bulk element type.

1.14 Delete interface elements

This function is identical to that in the two-dimensional problem.

1.15 Prescribing the initial free surface position

At the beginning of the simulation the interface is deformed by a prescribed function (9), implemented in the function `deform_free_surface(...)`, which cycles through the bulk mesh's `Nodes` and modifies their positions such that the nodes on the free surface follow the prescribed interface shape and the bulk nodes retain their fractional position between the solid boundaries and the (now deformed) interface.

```
//==start_of_deform_free_surface=======================================
/// Deform the mesh/free surface to a prescribed function
//======================================================================
template <class ELEMENT, class TIMESTEPPER>
void InterfaceProblem<ELEMENT,TIMESTEPPER>::
deform_free_surface(const double &epsilon,const double &k)
{
 // Initialise Bessel functions (only need the first!)
 double j0, j1, y0, y1, j0p, j1p, y0p, y1p;
 // Determine number of nodes in the "bulk" mesh
 const unsigned n_node = Bulk_mesh_pt->nnode();

 // Loop over all nodes in mesh
 for(unsigned n=0;n<n_node;n++)
  {
   // Determine eulerian position of node
   const double current_r_pos = Bulk_mesh_pt->node_pt(n)->x(0);
   const double current_z_pos = Bulk_mesh_pt->node_pt(n)->x(1);

   // Compute Bessel functions
   CRBond_Bessel::bessjy01a(k*current_r_pos,j0,j1,y0,y1,j0p,j1p,y0p,y1p);

   // Determine new vertical position of node
   const double new_z_pos = current_z_pos
    + (1.0-fabs(1.0-current_z_pos))*epsilon*j0;

   // Set new position
   Bulk_mesh_pt->node_pt(n)->x(1) = new_z_pos;
  }
} // End of deform_free_surface
```

1.16 Post-processing

This function follows an identical structure to that in the two-dimensional problem.

1.17 The timestepping loop

The function `unsteady_run(...)` is used to perform the timestepping procedure, and is very similar to that in the two-dimensional problem. The only difference arises from the fact that the initial interface deformation is a Bessel function rather than a cosine curve.

```
 // Set value of epsilon
```

```
const double epsilon = 0.1;

// Set value of k in Bessel function J_0(kr)
const double k_bessel = 3.8317;
// Deform the mesh/free surface
deform_free_surface(epsilon,k_bessel);
```

The rest of the function is identical to before.

1.18 Source files for this tutorial

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

  demo_drivers/axisym_navier_stokes/two_layer_interface_axisym/

- The driver code is:

  demo_drivers/axisym_navier_stokes/two_layer_interface_axisym/elastic_two_layer_interface_axisym.cc

1.19 PDF file

A pdf version of this document is available.