Chapter 1

Demo problem: Relaxation oscillations of a fluid layer

This is our first free surface example problem. We discuss the non-dimensionalisation of the free surface boundary conditions and their implementation in `oomph-lib`, and demonstrate the solution of a single layer relaxation problem.

1.1 Boundary conditions at a free surface

Free surfaces occur at the interface between two fluids. Such interfaces require two boundary conditions to be applied:

1. a kinematic condition which relates the motion of the free surface to the fluid velocities at the surface, and
2. a dynamic condition which is concerned with the force balance at the free surface.

1.1.1 The kinematic condition

The kinematic condition states that the fluid particles at the surface remain on the surface for all times. If the surface is parametrised by intrinsic coordinates ζ_1 and ζ_2, then the Eulerian position vector which describes the surface at a given time t can be written as $\mathbf{R}^* = \mathbf{R}^*(\zeta_1, \zeta_2, t)$. The kinematic condition is then given by

$$\left(u_i^* - \frac{\partial R_i^*}{\partial t^*}\right) n_i = 0,$$

where $\mathbf{u}^*$ is the (dimensional) velocity of the fluid and $\mathbf{n}$ is the outer unit normal to the free surface. Using the same problem-specific reference quantities for the velocity, $\mathcal{U}$, length, $\mathcal{L}$, and time, $\mathcal{T}$, that were used to non-dimensionalise the Navier-Stokes equations, we scale the dimensional quantities such that

$$u_i^* = \mathcal{U}\, u_i, \qquad R_i^* = \mathcal{L}\, R_i, \qquad t^* = \mathcal{T}\, t.$$

The non-dimensional form of the kinematic boundary condition is then given by

$$\left(u_i - St\, \frac{\partial R_i}{\partial t}\right) n_i = 0, \qquad (1)$$

where the Strouhal number is

$$St = \frac{\mathcal{L}}{\mathcal{U}\mathcal{T}}.$$

1.1.2 The dynamic condition

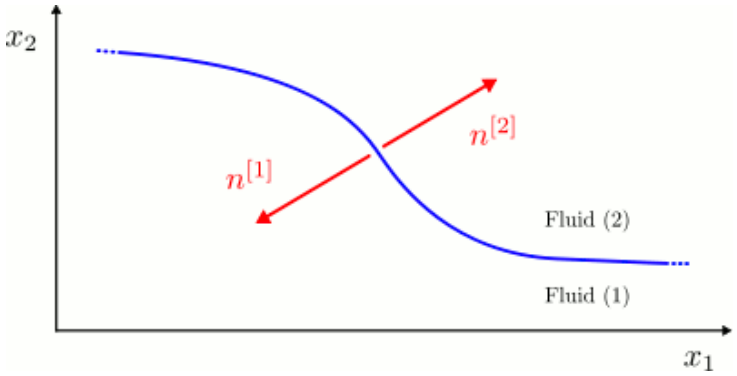

Figure 1.1 Sketch of the interface between two fluids.

The dynamic boundary condition requires the stress to be continuous across a flat interface between two fluids. Referring to the sketch above, we define the lower fluid to be fluid 1 and the upper fluid to be fluid 2. The traction exerted by fluid 1 onto fluid 2, $\mathbf{t}^{[1]*}$, is equal and opposite to that exerted by fluid 2 onto fluid 1, $\mathbf{t}^{[2]*}$, and therefore $\mathbf{t}^{[1]*} = -\mathbf{t}^{[2]*}$. The traction in fluid β ($\beta = 1, 2$) is given by

$$t_i^{[\beta]*} = \tau_{ij}^{[\beta]*} \, n_j^{[\beta]},$$

where $\tau^{[\beta]*}$ is the stress tensor in fluid β and $\mathbf{n}^{[\beta]}$ is the outer unit normal to fluid β. Since $\mathbf{n}^{[2]}$ must equal $-\mathbf{n}^{[1]}$, we have

$$\tau_{ij}^{[1]*} \, n_j^{[1]} = \tau_{ij}^{[2]*} \, n_j^{[1]},$$

where we have arbitrarily chosen to use $\mathbf{n}^{[1]}$ as the unit normal.
On curved surfaces, surface tension creates a pressure jump $\Delta p^* = \sigma \kappa^*$ across the interface, where σ is the surface tension and κ^* is equal to twice the mean curvature of the surface. Therefore the dynamic boundary condition is given by

$$\tau_{ij}^{[2]*} \, n_j^{[1]} = \tau_{ij}^{[1]*} \, n_j^{[1]} + \sigma \, \kappa^* \, n_i^{[1]},$$

where $\kappa > 0$ if the centre of curvature lies inside fluid

1. Using the same problem-specific reference quantities as in the section above, the dimensional quantities are scaled such that
$$\tau_{ij}^{*} = \frac{\mu_{ref} \, \mathcal{U}}{\mathcal{L}} \, \tau_{ij}, \qquad \kappa^* = \frac{1}{\mathcal{L}} \, \kappa.$$
The non-dimensional form of the dynamic boundary condition is then given by
$$\tau_{ij}^{[2]} \, n_j^{[1]} = \tau_{ij}^{[1]} \, n_j^{[1]} + \frac{1}{Ca} \, \kappa \, n_i^{[1]},$$
where the Capillary number is
$$Ca = \frac{\mu_{ref} \, \mathcal{U}}{\sigma}.$$

In certain cases, such as the example problem below, we wish to model the fluid above the interface as totally inviscid. In this case, the stress tensor in fluid 2 reduces to $\tau_{ij}^{[2]} = -\delta_{ij} p_{ext}$, where p_{ext} is the (non-dimensional) constant pressure above the free surface. The dynamic boundary condition therefore becomes

$$\tau_{ij} \, n_j = -\left(\frac{1}{Ca} \, \kappa + p_{ext} \right) n_i,$$

where we have dropped the explicit references to fluid 1 since it is understood that the stress tensor and unit normals refer to those of the (one and only) viscous fluid in the problem.
We shall now discuss how the free surface boundary conditions are implemented in `oomph-lib`.

1.2 Implementation

1.2.1 The kinematic condition and the pseudo-solid node-update procedure

In addition to solving for the fluid velocity and pressure (as in all Navier–Stokes examples), we have additional degrees of freedom in our problem due to the fact that the position of the free surface $\mathbf{R}$ is unknown. The Navier–Stokes elements in `oomph-lib` are based on the Arbitrary Lagrangian Eulerian (ALE) form of the Navier-Stokes equations, and so can be used to solve problems in moving domains. This allows us to discretise our domain using a boundary fitted mesh, which will need to deform in response to the motion of the free surface. This is achieved by treating the interior of the mesh as a fictitious elastic solid, and solving a solid mechanics problem for the (unknown) nodal positions. This technique, which will subsequently be referred to as a 'pseudo-solid node-update strategy', employs wrapper elements to existing fluid and solid equation classes. The specific element used in this example is a `PseudoSolidNodeUpdateElement<QCrouzeixRaviartElement<2>,QPVDElement<2,3>>` element, which takes two template arguments. The first is the standard element type used to solve the fluid problem, and the second is the element type which solves the equations that are used to control the mesh deformation. The deformation of the free surface boundary is imposed by introducing a field of Lagrange multipliers at the free surface, following the method outlined in Cairncross et al., 'A finite element method for free surface flows of incompressible fluids in three dimensions. Part I. Boundary fitted mesh motion' (2000). These new unknowns are stored as nodal values, and so the vector of values at each node is resized accordingly. Since this introduces further degrees of freedom into the problem, we require an additional equation: the kinematic boundary condition (1).
We discretise this equation by attaching `FaceElements` to the boundaries of the "bulk" elements that are adjacent to the free surface. The specific `FaceElement` used in this example is an `ElasticLineFluidInterfaceElement<ELEMENT>`, which takes the bulk element type as a template argument. This allows the user of the driver code to easily change the bulk element type, since the appropriate `FaceElement` type is automatically used. These `FaceElements` are applied in the same way as all other surface elements (e.g. `NavierStokesTractionElements`, `UnsteadyHeatFluxElements`, etc.), and a general introduction can be found in another tutorial.

1.2.2 The dynamic condition

Within a finite element framework, the dynamic boundary condition is incorporated as contributions to each of the momentum equations at the free surface. We refer to Ruschak, 'A method for incorporating free boundaries with surface tension in finite element fluid-flow simulators' (1980), for details on the formulation, which can also be found in our free surface theory document. Since both Taylor–Hood and Crouzeix–Raviart elements are implemented such that the normal stresses between elements are balanced, applying the dynamic boundary condition in cases in which we are solving the Navier–Stokes equations on both sides of the interface is as straightforward as adding the appropriate surface tension contributions to the relevant momentum equations at the interface. In cases such as the example below, where we have an inviscid fluid above the free surface, we need to add the appropriate external pressure contributions (if any) as well. Both of these contributions are automatically added to the appropriate momentum equations using the same `FaceElements` which are used to discretise the kinematic boundary condition (see above).
The Capillary number defaults to 1.0 and other values may be set using the function:
`double* FluidInterfaceElement::ca_pt().`
The Strouhal number defaults to 1.0 and other values may be set using the function:
`double* FluidInterfaceElement::st_pt().`
The external pressure defaults to zero and other values may be set using the function:
`void FluidInterfaceElement::set_external_pressure_data(Data* p_ext_data_pt),`
where `p_ext_data_pt` is (a pointer to) the `Data` in which the value of the external pressure is stored. We note that the external pressure is represented by `Data` because it may be an unknown in certain problems, although it is simply a constant parameter in the example below. It can be accessed using the function:
`double FluidInterfaceElement::pext().`
The way in which the dynamic condition is incorporated within our finite element structure is discussed in more detail in the comments at the end of this tutorial.

1.3 The example problem

We will illustrate the solution of the unsteady two-dimensional Navier–Stokes equations using the example of a distorted free surface which is allowed to relax. The domain is periodic in the x_1 direction.

The 2D unsteady Navier–Stokes equations under a distorted free surface.

Solve

$$Re\left(St\frac{\partial u_i}{\partial t} + u_j\frac{\partial u_i}{\partial x_j}\right) = -\frac{\partial p}{\partial x_i} + \frac{Re}{Fr}G_i + \frac{\partial}{\partial x_j}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \qquad (2)$$

and

$$\frac{\partial u_i}{\partial x_i} = 0, \qquad (3)$$

with gravity acting in the negative x_2 direction, in the unit square, where the free surface is located at $\mathbf{R}$, subject to the Dirichlet boundary conditions:

$$u_1 = 0 \qquad (4)$$

on the bottom, left and right boundaries and

$$u_2 = 0 \qquad (5)$$

on the bottom boundary.
The free surface is defined by $\mathbf{R}$, which is subject to the kinematic condition:

$$\left(u_i - St\,\frac{\partial R_i}{\partial t}\right) n_i = 0, \qquad (6)$$

and the dynamic condition:

$$\tau_{ij} n_j = -\left(\frac{1}{Ca}\kappa + p_{ext}\right) n_i, \qquad (7)$$

where the stress tensor is defined as:

$$\tau_{ij} = -p\,\delta_{ij} + \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right). \qquad (8)$$

The initial deformation of the free surface is defined by:

$$\mathbf{R} = x_1\,\mathbf{i} + [1.0 + \epsilon\cos(2n\pi x_1)]\,\mathbf{j} \qquad (9)$$

where ϵ is a small parameter and n is an integer.

1.4 Results

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

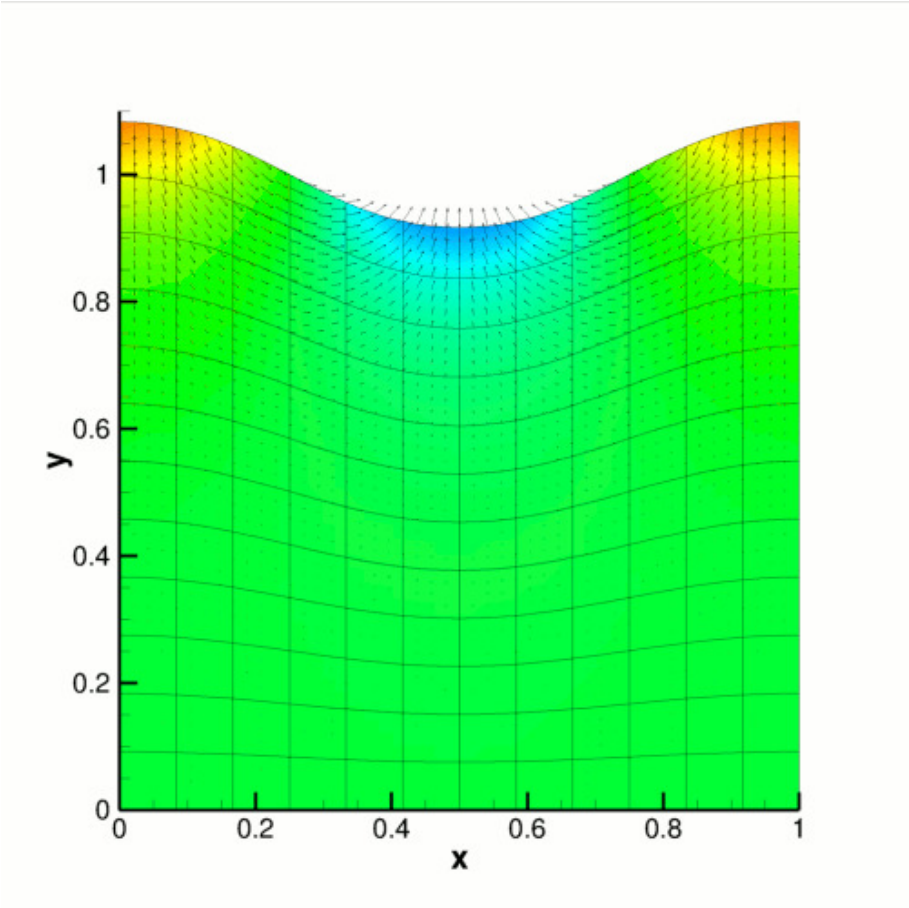

Figure 1.2 Pressure contour plot for the relaxing interface problem.

At time $t \leq 0$ the free surface is fixed in its deformed shape, but as the simulation begins the restoring forces of surface tension and gravitational acceleration act to revert it to its undeformed flat state. The surface 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. This viscous damping effect can be seen in the following time-trace of the height of the fluid layer at the edge of the domain.

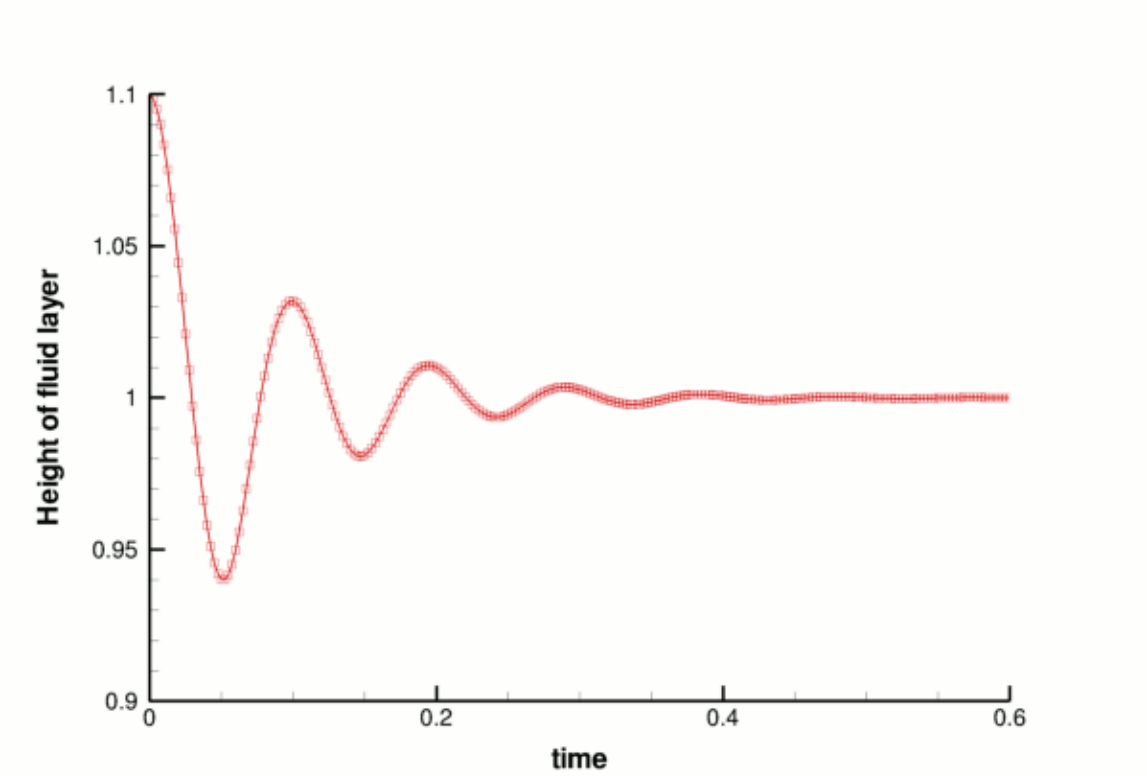

Figure 1.3 Time-trace of the height of the fluid layer at the edge of the domain.

1.5 Validation

The free surface boundary conditions for the Cartesian 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 by proposing that we can write the fluid velocities and pressure, $u(x,y,t)$, $v(x,y,t)$ and $p(x,y,t)$, as well as the 'height' of the interface, $h(x,t)$, in the form $x = \bar{x} + \epsilon\hat{x}$, where the barred quantities correspond to the 'base' state, chosen here to be the trivial solution $\bar{h} = \bar{u} = \bar{v} = \bar{p} = 0$. Once the linearised forms of the governing equations and boundary conditions have been determined we propose a separable solution of the form

$$\hat{h}(x,t) = He^{\lambda t + ikx},$$
$$\hat{u}(x,y,t) = U(y)e^{\lambda t + ikx},$$
$$\hat{v}(x,y,t) = V(y)e^{\lambda t + ikx},$$

and

$$\hat{p}(x,y,t) = P(y)e^{\lambda t + ikx}.$$

Substituting the above ansatz into the governing equations results in a system of coupled ordinary differential equations for $U(y)$, $V(y)$ and $P(y)$ which we solve to find their general solutions up to a set of unknown constants A, B, C and D. By substituting these general forms into the set of (linearised and separated) boundary conditions we obtain a linear system of five equations in the five unknowns A, B, C, D and H, from which we can assemble a homogeneous linear system of the form

$$\mathbf{M}\begin{bmatrix} A \\ B \\ C \\ D \\ H \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where $\mathbf{M}$ is a 5×5 matrix whose entries are the coefficients of the unknowns in our five conditions. This system only has a non-trivial solution if $|\mathbf{M}| = 0$, and solving this equation gives us λ as a function of k. This is a dispersion

relation and describes how wave propagation varies as a function of its wavenumber. More specifically, the real part of λ is the growth rate of the wave and the imaginary part is its frequency. This analytical result can now 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.

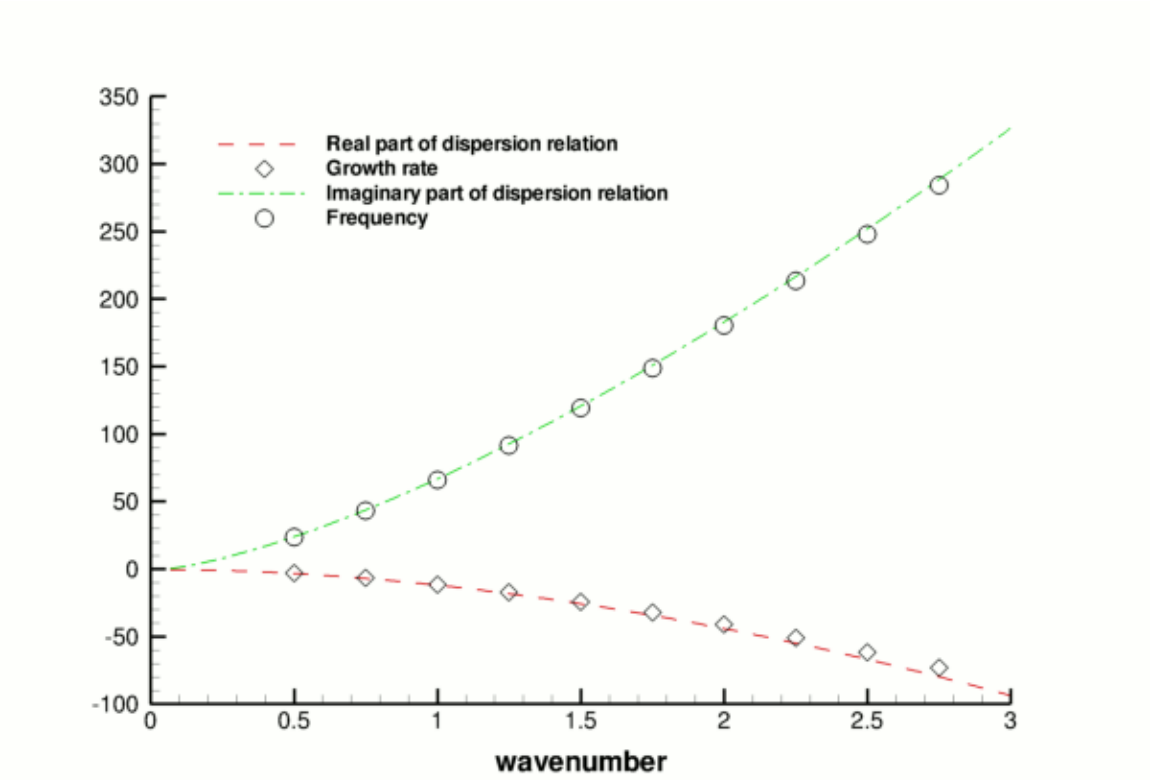

Figure 1.4 Validation of the code (points) by comparison with an analytical dispersion relation (lines).

1.6 Global parameters and functions

As usual, we use a namespace to define the dimensionless parameters Re, St, Re/Fr and Ca, and we create a vector G which will define the direction in which gravity acts. We will need to pass the Strouhal number to the interface elements, but the product of the Strouhal number and the Reynolds number to the bulk elements. To avoid potentially inconsistent parameters, we compute $Re\,St$ rather than defining it explicitly. Because the mesh is to be updated using a pseudo-solid node-update strategy, we also require the Poisson ratio for the generalised Hookean constitutive law.

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

 /// Reynolds number
 double Re = 5.0;

 /// Strouhal number
 double St = 1.0;

 /// Womersley number (Reynolds x Strouhal, computed automatically)
 double ReSt;

 /// Product of Reynolds number and inverse of Froude number
 double ReInvFr = 5.0; // (Fr = 1)

 /// Capillary number
 double Ca = 0.01;

 /// Direction of gravity
 Vector<double> G(2);
```

```
 /// Pseudo-solid Poisson ratio
 double Nu = 0.1;

} // End of namespace
```

1.7 The driver code

We start by computing the product of the Reynolds and Strouhal numbers before specifying the (non-dimensional) length of time for which we want the simulation to run 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 two-dimensional single fluid free surface problem
//======================================================================
int main(int argc, char* argv[])
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);

 // Compute the Womersley number
 Global_Physical_Variables::ReSt =
  Global_Physical_Variables::Re*Global_Physical_Variables::St;

 /// Maximum time
 double t_max = 0.6;

 /// Duration of timestep
 const double dt = 0.0025;

 // If we are doing validation run, use smaller number of timesteps
 if(CommandLineArgs::Argc>1) { t_max = 0.005; }
```

Next we specify the dimensions of the mesh and the number of elements in the x_1 and x_2 directions. To remain consistent with the example code we shall from now on refer to x_1 as x and x_2 as y.

```
 // Number of elements in x direction
 const unsigned n_x = 12;

 // Number of elements in y direction
 const unsigned n_y = 12;

 // Width of domain
 const double l_x = 1.0;

 // Height of fluid layer
 const double h = 1.0;
```

At this point we define the direction in which gravity acts: vertically downwards.

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

Finally, we build the problem using the 'pseudo-solid' version of `QCrouzeixRaviartElements` 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 QCrouzeixRaviartElements,
 // using the BDF<2> timestepper
 InterfaceProblem<PseudoSolidNodeUpdateElement< QCrouzeixRaviartElement<2>,
  QPVDElement<2,3> > , BDF<2> >
  problem(n_x,n_y,l_x,h);
 // Run the unsteady simulation
 problem.unsteady_run(t_max,dt);
} // End of main
```

1.8 The problem class

Since we are solving the unsteady Navier–Stokes equations, the `Problem` class 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, before passing 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============================================
/// Single fluid free surface problem in a rectangular domain which is
/// periodic in the x direction
//====================================================================
template <class ELEMENT, class TIMESTEPPER>
class InterfaceProblem : public Problem
{
public:
```

```
 /// Constructor: Pass the number of elements and the lengths of the
 /// domain in the x and y directions (h is the height of the fluid layer
 /// i.e. the length of the domain in the y direction)
 InterfaceProblem(const unsigned &n_x, const unsigned &n_y,
                  const double &l_x, const double &h);

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

 /// 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);
```

The nodal positions are unknowns in the problem and hence are updated automatically, so there is no need to update the mesh before performing a Newton solve. However, since the main use of the methodology demonstrated here is in free-boundary problems where the solution of the solid problem merely serves to update the nodal positions in response to the motion of the free surface, we reset the nodes' Lagrangian coordinates to their Eulerian positions before every solve, by calling `SolidMesh::set_lagrangian_nodal_coordinates()`. This makes the deformed configuration stress-free and tends to stabilise the computation, allowing larger domain deformations to be computed.

```
private:

 /// No actions required before solve step
 void actions_before_newton_solve() {}

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

 /// Actions before the timestep: For maximum stability, reset
 /// the current nodal positions to be the "stress-free" ones.
 void actions_before_implicit_timestep()
  {
   Bulk_mesh_pt->set_lagrangian_nodal_coordinates();
  }

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

The problem class stores pointers to the specific bulk mesh and the surface mesh, which will contain the interface elements, as well as a pointer to a constitutive law for the pseudo-solid mesh. The width of the domain is also stored since it is used by the function `deform_free_surface(...)` when setting up the initial mesh deformation. Finally we store an output stream in which we record the height of the interface at the domain edge.

```
 /// Pointer to the (specific) "bulk" mesh
 ElasticRectangularQuadMesh<ELEMENT>* Bulk_mesh_pt;

 /// Pointer to the "surface" mesh
 Mesh* Surface_mesh_pt;

 // Pointer to the constitutive law used to determine the mesh deformation
 ConstitutiveLaw* Constitutive_law_pt;

 /// Width of domain
 double Lx;

 /// Trace file
 ofstream Trace_file;

}; // End of problem class
```

1.9 The problem constructor

The constructor starts by copying the width of the domain into the private member data of the problem class, before building the timestepper.

```
//==start_of_constructor=============================================
/// Constructor for single fluid free surface problem
//====================================================================
template <class ELEMENT, class TIMESTEPPER>
InterfaceProblem<ELEMENT,TIMESTEPPER>::
InterfaceProblem(const unsigned &n_x, const unsigned &n_y,
                 const double &l_x, const double& h) : Lx(l_x)
```

```
{

 // Allocate the timestepper (this constructs the time object as well)
 add_time_stepper_pt(new TIMESTEPPER);
```

Next we build the bulk mesh. The mesh we are using is the `ElasticRectangularQuadMesh<ELEMENT>`, which takes the bulk element as a template argument. The boolean argument in the mesh constructor, which is set to 'true' here, indicates whether or not the domain is to be periodic in x. The surface mesh is also built, although it is empty at this point.

```
 // Build and assign the "bulk" mesh (the "true" boolean flag tells
 // the mesh constructor that the domain is periodic in x)
 Bulk_mesh_pt = new ElasticRectangularQuadMesh<ELEMENT>
  (n_x,n_y,l_x,h,true,time_stepper_pt());

 // Create the "surface mesh" that will contain only the interface
 // elements. The constructor just creates the mesh without giving
 // it any elements, nodes, etc.
 Surface_mesh_pt = new Mesh;
```

Having created the bulk elements, we now create the interface elements. We first build an empty mesh in which to store them, before looping over the bulk elements adjacent to the free surface and 'attaching' interface elements to their upper faces. These newly-created elements are then stored in the surface mesh.

```
 // Create the interface elements
 // -----------------------------

 // Loop over those elements adjacent to the free surface
 for(unsigned e=0;e<n_x;e++)
  {
   // Set a pointer to the bulk element we wish to our interface
   // element to
   FiniteElement* bulk_element_pt =
    Bulk_mesh_pt->finite_element_pt(n_x*(n_y-1)+e);

   // Create the interface element (on face 2 of the bulk element)
   FiniteElement* interface_element_pt =
    new ElasticLineFluidInterfaceElement<ELEMENT>(bulk_element_pt,2);

   // Add the interface element to the surface mesh
   this->Surface_mesh_pt->add_element_pt(interface_element_pt);
  }
```

Now that the interface elements have been created, we combine the bulk and surface meshes into a single mesh.

```
 // Add the two sub-meshes to the problem
 add_sub_mesh(Bulk_mesh_pt);
 add_sub_mesh(Surface_mesh_pt);

 // Combine all sub-meshes into a single mesh
 build_global_mesh();
```

On the solid bottom boundary ($y = 0$) we pin both velocity components so that there is no penetration of the wall by the fluid or flow along it. On the left and right symmetry boundaries ($x = 0.0$ and $x = 1.0$) we pin the x component of the velocity but leave the y component unconstrained. We do not apply any velocity boundary conditions to the free surface (the top boundary). We pin the vertical displacement of the nodes on the bottom boundary (since these must remain stationary) and pin the horizontal displacement of all nodes in the mesh.

```
 // 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:
     // --------------------------

     // On lower boundary (solid wall), pin x and y components of
     // the velocity (no slip/penetration)
     if(b==0)
      {
       Bulk_mesh_pt->boundary_node_pt(b,n)->pin(0);
       Bulk_mesh_pt->boundary_node_pt(b,n)->pin(1);
      }

     // On left and right boundaries, pin x-component of the velocity
     // (no penetration of the periodic boundaries)
     if(b==1 || b==3)
```

```
      {
       Bulk_mesh_pt->boundary_node_pt(b,n)->pin(0);
      }

     // Solid boundary conditions:
     // --------------------------

     // On lower boundary (solid wall), pin vertical displacement
     // (no penetration)
     if(b==0)
      {
       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

 // Pin horizontal displacement of all nodes
 const unsigned n_node = Bulk_mesh_pt->nnode();
 for(unsigned n=0;n<n_node;n++) { Bulk_mesh_pt->node_pt(n)->pin_position(0); }
```

Next we create a generalised Hookean constitutive equation for the pseudo-solid mesh. This constitutive equation is discussed in another tutorial.

```
 // Define a constitutive law for the solid equations: generalised Hookean
 Constitutive_law_pt = new GeneralisedHookean(&Global_Physical_Variables::Nu);
```

We loop over the bulk elements and pass them 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, and the constitutive law. In addition we pass a pointer to the global time object, created when we called `Problem::add_time_stepper_pt(...)` above.

```
 // Complete the problem setup to make the elements fully functional
 // ----------------------------------------------------------------
 // Determine number of bulk elements in mesh
 const unsigned n_element_bulk = Bulk_mesh_pt->nelement();

 // Loop over the bulk elements
 for(unsigned e=0;e<n_element_bulk;e++)
  {
   // Upcast from GeneralisedElement to the present element
   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(Bulk_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;

   // Set the product of the Reynolds number and the inverse of the
   // Froude number
   el_pt->re_invfr_pt() = &Global_Physical_Variables::ReInvFr;

   // Set the direction of gravity
   el_pt->g_pt() = &Global_Physical_Variables::G;

   // Set the constitutive law
   el_pt->constitutive_law_pt() = Constitutive_law_pt;

  } // End of loop over bulk elements
```

Next we create a pointer to a `Data` value for the external pressure p_{ext}, before pinning it and assigning an arbitrary value.

```
 // Create a Data object whose single value stores the external pressure
 Data* external_pressure_data_pt = new Data(1);
 // Pin and set the external pressure to some arbitrary value
 external_pressure_data_pt->pin(0);
 external_pressure_data_pt->set_value(0,1.31);
```

We then loop over the interface elements and pass them a pointer to this external pressure value as well as pointers to the Strouhal and Capillary numbers.

```
 // Determine number of 1D interface elements in mesh
 const unsigned n_interface_element = Surface_mesh_pt->nelement();

 // Loop over the interface elements
 for(unsigned e=0;e<n_interface_element;e++)
  {
   // Upcast from GeneralisedElement to the present element
   ElasticLineFluidInterfaceElement<ELEMENT>* el_pt =
    dynamic_cast<ElasticLineFluidInterfaceElement<ELEMENT>*>
    (Surface_mesh_pt->element_pt(e));

   // Set the Strouhal number
   el_pt->st_pt() = &Global_Physical_Variables::St;

   // Set the Capillary number
   el_pt->ca_pt() = &Global_Physical_Variables::Ca;

   // Pass the Data item that contains the single external pressure value
```

```
   el_pt->set_external_pressure_data(external_pressure_data_pt);

  } // End of loop over interface elements
```

Finally, we apply the problem's boundary conditions (discussed later on) before setting up the equation numbering scheme using the function `Problem::assign_eqn_numbers()`.

```
 // Apply the boundary conditions
 set_boundary_conditions();

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

} // End of constructor
```

1.10 Initial conditions

This function sets the initial conditions for the problem. We loop over all nodes in the mesh and set both 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, as well as the current nodal positions, into the required number of history values for the timestepper in question. This corresponds to an impulsive start, as for all time $t \leq 0$ none of the fluid is moving and the shape of the interface is constant.

```
//==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 = mesh_pt()->nnode();
 // Loop over all nodes in mesh
 for(unsigned n=0;n<n_node;n++)
  {
   // Loop over the two velocity components
   for(unsigned i=0;i<2;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 and nodal positions
 // for timestepping corresponding to an impulsive start
 assign_initial_values_impulsive();
} // End of set_initial_condition
```

1.11 Boundary conditions

This function sets the boundary conditions for the problem. Since the Dirichlet conditions are homogeneous this function is not strictly necessary as all values are initialised to zero by default.

```
//==start_of_set_boundary_conditions=====================================
/// 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 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 x-component of the velocity to zero on all boundaries
     // other than the free surface
     if(b!=2)
      {
       Bulk_mesh_pt->boundary_node_pt(b,n)->set_value(0,0.0);
      }

     // Set y-component of the velocity to zero on the bottom wall
     if(b==0)
      {
```

```
       Bulk_mesh_pt->boundary_node_pt(b,n)->set_value(1,0.0);
      }
    } // End of loop over nodes on boundary b
  } // End of loop over mesh boundaries
} // End of set_boundary_conditions
```

1.12 Prescribing the initial free surface position

At the beginning of the simulation the free surface is deformed by a prescribed function (9). To do this we define a function, `deform_free_surface(...)`, which cycles through the bulk mesh's `Nodes` and modifies their positions accordingly, such that the nodes on the free surface follow the prescribed interface shape (9) and the bulk nodes retain their fractional position between the lower and the (now deformed) upper boundary.

```
//==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 unsigned &n_periods)
{
 // 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_x_pos = Bulk_mesh_pt->node_pt(n)->x(0);
   const double current_y_pos = Bulk_mesh_pt->node_pt(n)->x(1);

   // Determine new vertical position of node
   const double new_y_pos = current_y_pos
    + (1.0-fabs(1.0-current_y_pos))*epsilon
    *(cos(2.0*n_periods*MathematicalConstants::Pi*current_x_pos/Lx));

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

1.13 Post-processing

As expected, this member function documents the computed solution. We first output the value of the current time to the screen, before recording the continuous time and the height of the free surface at the domain boundary in the trace file. We note that as the domain is periodic the height of the free surface must be the same at both the left and right boundaries.

```
//==start_of_doc_solution=================================================
/// Document the solution
//========================================================================
template <class ELEMENT, class TIMESTEPPER>
void InterfaceProblem<ELEMENT,TIMESTEPPER>::doc_solution(DocInfo &doc_info)
{

 // Output the time
 cout « "Time is now " « time_pt()->time() « std::endl;

 // Upcast from GeneralisedElement to the present element
 ElasticLineFluidInterfaceElement<ELEMENT>* el_pt =
  dynamic_cast<ElasticLineFluidInterfaceElement<ELEMENT>*>
  (Surface_mesh_pt->element_pt(0));

 // Document time and vertical position of left hand side of interface
 // in trace file
 Trace_file « time_pt()->time() « " "
            « el_pt->node_pt(0)->x(1) « std::endl;
```

We then output the computed solution.

```
 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
 Bulk_mesh_pt->output(some_file,npts);

 // Close solution output file
 some_file.close();
```

Finally, we output the shape of the interface.

```
 // Open interface solution output file
 sprintf(filename,"%s/interface_soln%i.dat",
         doc_info.directory().c_str(),doc_info.number());
 some_file.open(filename);
 // Output solution to file
 Surface_mesh_pt->output(some_file,npts);
 // Close solution output file
 some_file.close();
} // End of doc_solution
```

1.14 The timestepping loop

The function `unsteady_run(...)` is used to perform the timestepping procedure. We start by deforming the free surface in the manner specified by equation (9).

```
//==start_of_unsteady_run=============================================
/// Perform run up to specified time t_max with given timestep dt
//====================================================================
template <class ELEMENT, class TIMESTEPPER>
void InterfaceProblem<ELEMENT,TIMESTEPPER>::
unsteady_run(const double &t_max, const double &dt)
{

 // Set value of epsilon
 const double epsilon = 0.1;

 // Set number of periods for cosine term
 const unsigned n_periods = 1;

 // Deform the mesh/free surface
 deform_free_surface(epsilon,n_periods);
```

We then create a `DocInfo` object to store the output directory and the label for the output files.

```
 // Initialise DocInfo object
 DocInfo doc_info;

 // Set output directory
 doc_info.set_directory("RESLT");

 // Initialise counter for solutions
 doc_info.number()=0;
```

Next we open and initialise the trace file.

```
 // Open trace file
 char filename[100];
 sprintf(filename,"%s/trace.dat",doc_info.directory().c_str());
 Trace_file.open(filename);

 // Initialise trace file
 Trace_file << "time, free surface height" << std::endl;
```

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 conditions
 set_initial_condition();
```

We determine the number of timesteps to be performed and document the initial conditions, and then perform the actual timestepping loop. For each timestep the function `unsteady_newton_solve(dt)` is called and the solution documented.

```
 // 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()++;

 // Timestepping loop
 for(unsigned t=1;t<=n_timestep;t++)
  {
   // Output current timestep to screen
   cout << "\nTimestep " << t << " of " << n_timestep << std::endl;

   // Take one fixed timestep
   unsteady_newton_solve(dt);

   // Doc solution
```

```
   doc_solution(doc_info);

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

  } // End of timestepping loop

} // End of unsteady_run
```

1.15 Comments

1.15.1 The application of the dynamic boundary condition within the FEM

As discussed in an earlier tutorial, the finite element solution of the Navier–Stokes equations is based on their weak form, which is obtained by weighting the stress-divergence form of the momentum equations with the global test functions ψ_l, and integrating by parts to obtain the discrete residuals

$$f_{il} = \int_D \left[Re\left(St\frac{\partial u_i}{\partial t} + u_j\frac{\partial u_i}{\partial x_j} - \frac{G_i}{Fr}\right)\psi_l + \tau_{ij}\frac{\partial \psi_l}{\partial x_j}\right] dV - \int_{\partial D} \tau_{ij}\, n_j\, \psi_l\, dS = 0. \qquad (10)$$

Weighting the dynamic condition (7) by the same global test functions ψ_l and integrating over the domain boundary ∂D gives

$$\int_{\partial D} \tau_{ij} n_j\, \psi_l\, dS = -\int_{\partial D} p_{ext} n_i\, \psi_l\, dS - \int_{\partial D} \frac{1}{Ca}\kappa\, \psi_l\, dS. \qquad (11)$$

In a two-dimensional problem, such as the one considered in this tutorial, the domain boundary reduces to a one-dimensional curve, C. A further integration by parts of (11) therefore gives

$$\int_{\partial D} \tau_{ij} n_j\, \psi_l\, dS = -\int_{C} p_{ext} n_i\, \psi_l\, dS + \int_C \frac{1}{Ca} t_i\, \frac{\partial \psi_l}{\partial S}\, dS - \frac{1}{Ca}\left[t_i\, \psi_l\right]_{c_1}^{c_2}, \qquad (12)$$

where t_i, $(i = 1, 2)$ is the i-th component of a unit vector tangent to C and pointing in the direction of increasing S, and c_1 and c_2 are the two endpoints of C.
In the problem considered in this tutorial, the domain boundary C can be written as $C = C_{solid} \cup C_{surface}$, where C_{solid} represents the portion of the domain boundary corresponding to a rigid wall and $C_{surface}$ represents the portion corresponding to the free surface. The velocity along C_{solid} is prescribed by Dirichlet boundary conditions, and we recall that the global test functions ψ_l vanish in this case. On non-Dirichlet boundaries we must either specify the external pressure p_{ext}, or deliberately neglect this term to obtain the 'natural' condition $p_{ext} = 0$.

1.15.2 The contact line

In this two-dimensional case, the contact 'line' actually reduces to two contact points, c_1 and c_2, which are located at either side of the portion of the domain boundary corresponding to the free surface $C_{surface}$. The two point contributions are added by specifying the the tangent to the surface t_i at each of the contact points, which is equivalent to prescribing the contact angle that the free surface makes with the neighbouring domain boundary. We note that in the problem considered here we do not explicitly apply any boundary conditions at either end of the free surface. Neglecting these contributions corresponds to the 'natural' condition of prescribing a 90^o contact angle, which happens to be the appropriate condition in this case. Contact angles of arbitrary size can be enforced using `FluidInterfaceBoundingElements`, which are discussed in a later tutorial.
Specifying the contact angle is not the only condition that can be applied at the edges of an interface. The alternative boundary condition is to pin the contact line so that its position is fixed for all time. Since this is a Dirichlet condition it causes the integral over the contact line to vanish.

1.16 Source files for this tutorial

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

  demo_drivers/navier_stokes/single_layer_free_surface/

- The driver code is:

  demo_drivers/navier_stokes/single_layer_free_surface/elastic_single_↩
  layer.cc

1.17 PDF file

A pdf version of this document is available.