Chapter 1

A fluid-structure-interaction problem: Flow in a 2D collapsible channel

In this document we discuss the solution of a standard fluid-structure interaction problem – finite-Reynolds-number flow in a 2D collapsible channel. We shall demonstrate that the driver code for this multi-physics problem is a straightforward combination of the driver codes for the two corresponding single-physics problems discussed earlier:

- **Flow in a 2D channel with prescribed wall motion:**

  In this single-physics problem, we represented the moving wall by a `GeomObject`, and created a `Domain` object to provide
  an analytical representation of the domain boundary. We discretised the Navier-Stokes equations with 2D Crouzeix-Raviart elements and updated their nodal positions (in response to the prescribed changes in the wall position) by their `MacroElement` representation.

and

- **The deformation of a pressure-loaded elastic beam:**

  In this single-physics problem, we discretised the elastic beam with `HermiteBeamElements` and computed its deformation (determined by the positional degrees of freedom, stored at the `HermiteBeam↩ Element's SolidNodes`) in response to the prescribed pressure load.

1.1 The problem

Flow in a 2D collapsible channel

The figure below shows a sketch of the problem: Flow is driven by a prescribed pressure drop through a 2D channel of width H^* and total length $L^*_{total} = L^*_{up} + L^*_{collapsible} + L^*_{down}$. The upstream and downstream lengths of the channel are rigid, whereas the upper wall in the central section is an elastic membrane whose shape is parametrised by a Lagrangian coordinate, ξ^* , so that the position vector to the moving wall is given by $\mathbf{R}^*_w(\xi^*, t^*)$. The wall is loaded by the external pressure p^*_{ext} and by the traction that the viscous fluid exerts on it. The components of the load vector $\mathbf{f}^*$ that acts on the wall are therefore given by

$$f^*_i = (p^* - p^*_{ext})N_i - \mu \left(\frac{\partial u^*_i}{\partial x^*_j} + \frac{\partial u^*_j}{\partial x^*_i} \right) N_j \qquad \text{for } i = 1, 2,$$

where N_i (for $i = 1, 2$) are the components of the outer unit normal on the fluid domain.

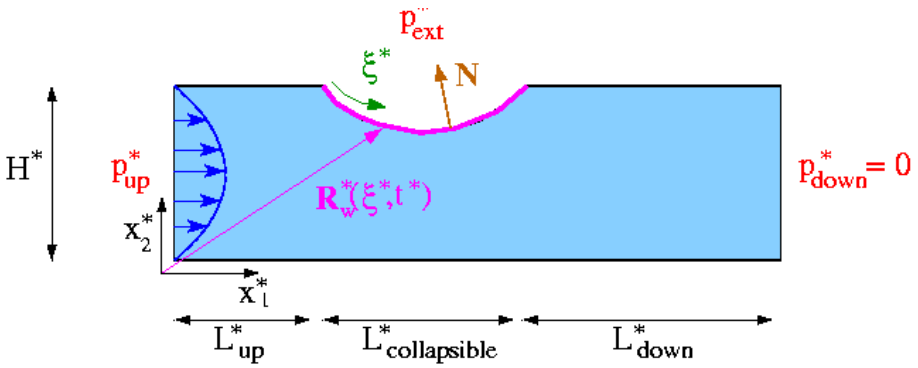

Figure 1.1 Sketch of the problem.

We scale all lengths on the channel width, H^* , use the average velocity through the undeformed channel, $U = P^*_{up} H^{*2}/(12\mu L^*_{total})$, to scale the velocities, and use H^*/U to non-dimensionalise time. Finally, the fluid pressure is non-dimensionalised on the viscous scale $p^* = p\mu U/H^*$. (As usual, asterisks distinguish dimensional parameters from their non-dimensional equivalents.)
With this non-dimensionalisation, the Navier-Stokes equations have the same form as in the earlier example with prescribed wall motion:

$$Re \left(St \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \qquad (1)$$

and

$$\frac{\partial u_i}{\partial x_i} = 0, \qquad (2)$$

with $St = 1$. As before, the flow is subject to the following boundary and initial conditions:

- Initial condition: Poiseuille flow, i.e.

$$\mathbf{u}(x_1, x_2, t = 0) = \mathbf{u}_{Poiseuille}(x_1, x_2) = 6\, x_2\, (1 - x_2)\, \mathbf{e}_1. \qquad (3)$$

- Parallel inflow, $\mathbf{u} \cdot \mathbf{e}_2 = \mathbf{0}$, and an applied axial traction of $\mathbf{t} \cdot \mathbf{e}_1 = p_{up} = 12\, L_{total}$ at the upstream end, $x_1 = 0$.
- Parallel, axially traction-free outflow at the downstream end, i.e. $\mathbf{u} \cdot \mathbf{e}_2 = \mathbf{0}$ and $\mathbf{t} \cdot \mathbf{e}_1 = p_{down} = 0$ at $x_1 = L_{total}$.
- No slip on all channel walls, i.e. $\mathbf{u} = \mathbf{0}$ on the rigid walls and

$$\mathbf{u} = \frac{\partial \mathbf{R}_w}{\partial t} \quad \text{on the moving wall,} \qquad (4)$$

These boundary conditions are identical to those in the problem with prescribed wall motion, apart from the fact that in the present problem the wall motion, described by $\mathbf{R}_w(\xi, t)$, has to be determined as part of the solution.

We model the elastic membrane as a thin-walled elastic Kirchhoff-Love beam of wall thickness h^* , subject to an axial (2nd Piola-Kirchhoff) pre-stress σ^*_0. The beam' effective (1D) elastic modulus is given by $E_{eff} = E/(1 - \nu^2)$, where E and ν are its 3D Young's modulus and Poisson's ratio, respectively. The beam's deformation is governed

1.2 Results

The figure below shows a snapshot of the flow field, taken from the animation of the computational results. The first four figures show (from top left to bottom right) "carpet plots" of the axial and transverse velocities, the axial component of the perturbation velocity $\mathbf{u} - \mathbf{u}_{Poiseuille}$, and the pressure distribution. The 2D contour plot at the bottom of the figure shows a contour plot of the pressure and a few instantaneous streamlines. The overall structure of the flow field is very similar to that observed in the corresponding problem with prescribed wall motion: The wall oscillation generates a large-amplitude sloshing flow that is superimposed on the pressure-driven Poiseuille flow. At the instant shown in this figure, the wall is moving inwards. Consequently, the sloshing flow generated in the upstream (downstream) rigid sections is directed against (in the same)
direction as that of the pressure-driven mean flow.

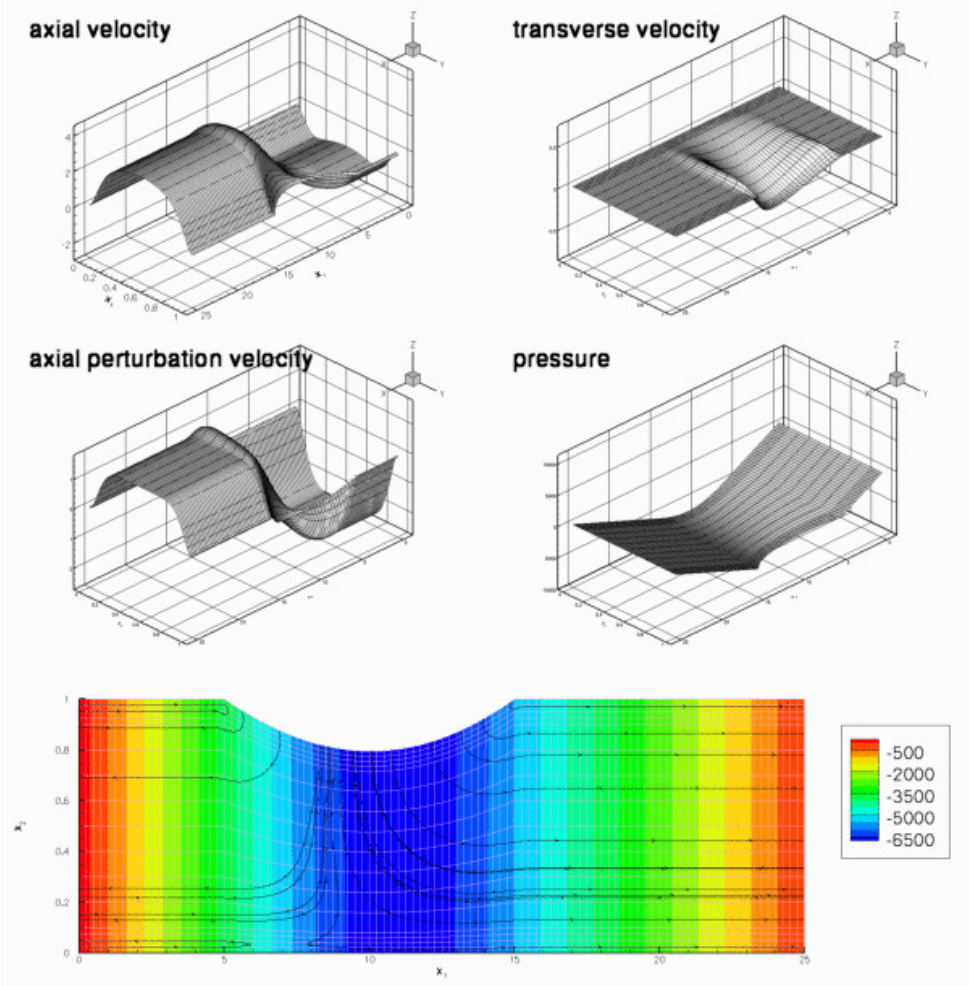

Figure 1.2 Snapshot from the animation of the flow field.

In the present problem, the wall motion is, of course, not prescribed but determined as part of the overall solution. Jensen & Heil's (2003) asymptotic analysis of the problem shows that the period of the oscillations is determined by the balance between fluid inertia and the elastic restoring forces. In certain parameter regimes (at sufficiently large Reynolds number), the wall can extract energy from the pressure-driven mean flow, causing the oscillations to grow in amplitude. In the present example, the Reynolds number is too small for this to happen and viscous dissipation causes the oscillations to decay, as shown in this plot.

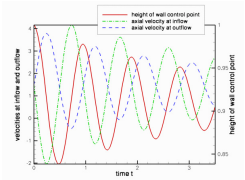

Figure 1.3 Time-trace of the axial velocities at two control points in the upstream and downstream cross-sections, and the vertical position of a control point on the wall.

1.3 Overview: How to solve fluid-structure interaction problems with oomph-lib

Before attempting to solve a fluid-structure interaction problem, we generally recommend to first study the constituent single-physics problems in isolation, i.e.

- Discretise the solid mechanics problem with a suitable `SolidFiniteElement` (e.g. a `HermiteBeam↩Element`) and determine the wall deformation in response to a prescribed external load. If possible, choose a load that is vaguely "representative" of the loads expected in the actual fluid-structure interaction problem.
- Discretise the fluid mechanics problem in a domain in which the deformation of the "elastic boundary" is described by a `GeomObject` that performs a prescribed motion. Choose a suitable node-update strategy to adjust the position of the fluid nodes in response to the (prescribed) motion of the "elastic boundary". (As demonstrated in many previous examples, a `Domain/MacroElement` - based node-update method is very easy to implement.) If possible, try to impose a wall motion that is "representative" of the type of wall motion expected in the actual fluid-structure interaction problem.

Once the behaviour of the two isolated single-physics problems is sufficiently well understood (e.g. what spatial discretisations are required, etc.), combine the two single-physics problems to a fully-coupled problem. The coupling introduces two types of interactions that must be incorporated into the computational framework:

1. The position of the nodes in the fluid mesh depends on the wall shape. Since the wall shape is now determined by the degrees of freedom in the `SolidFiniteElements` that we use to discretise the wall, we must ensure that:
   (a) The node-update operations that we developed for problems with prescribed wall motion (in which the position of the curvilinear domain boundaries is determined by `GeomObjects`) also work for problems in which the curvilinear domain boundaries are represented by `SolidFiniteElements`.
   (b) The dependence of the residuals of the fluid elements on the solid-mechanics degrees of freedom that affect the positions of the fluid nodes (via the node-update operation) are taken into account when computing the fluid elements' Jacobian matrices. (See the discussion of the "shape derivatives" in the `"toy" free-boundary Poisson problem` for details.)

2. The `SolidFiniteElements` are not only loaded by the external load but also by the traction that the fluid exerts on them. When computing the residuals of the `SolidFiniteElements` we have to evaluate the combined load vector at the `SolidFiniteElement's` Gauss points. We therefore have to provide a lookup scheme that specifies:
   (a) which fluid element is adjacent to a given Gauss point in the `SolidFiniteElements`, and
   (b) which (fluid) degrees of freedom affect the fluid traction at that point.

   The dependence of the residuals of the `SolidFiniteElements` on these (fluid) degrees of freedom must be taken into account when computing the `SolidFiniteElement's` Jacobian matrix.

`oomph-lib` provides a number of high-level functions that allow the required lookup schemes to be generated completely automatically. In the following sections we shall provide a brief discussion of the methodology but we stress that the details are not particularly important for the "user". If you just want to "use" `oomph-lib's` fluid-structure interaction capabilities and don't care too much about the technical details, you may wish to skip the next few sections and continue with the Overview of the driver code, where we demonstrate that, apart from a few trivial modifications, the driver code for the fully-coupled fluid-structure interaction problem is a straightforward combination of the two single-physics driver codes.

1.4 Brief discussion of the implementation

1.4.1 MacroElement-based (fluid-)node updates in FSI problems

1.4.1.1 The shape derivatives

When discussing our "toy" free-boundary Poisson problem we demonstrated how a "bulk" element's MacroElement - representation allows the efficient automatic evaluation of the "shape derivatives" – the derivatives of the "bulk" equations (here the Navier-Stokes equations) with respect to the Data values (here the nodal positions in the beam elements) that determine the shape of the domain boundary. The methodology discussed in the context of the "toy" problem may also be used for genuine fluid-structure interaction problems, such as the problem considered here, provided

1. A MacroElement/Domain - based Mesh is used to discretise the fluid domain.
2. The moving boundary is represented by a GeomObject whose member function GeomObject::geom↩_data_pt(...) provides access to its "geometric" Data, i.e. the Data that affects its shape.

Condition 1 is satisfied as the CollapsibleChannelMesh used for the simulation of the single-physics fluids problem with prescribed wall motion employs the CollapsibleChannelDomain to perform the node update in response to changes in the domain boundary. Multiple inheritance may therefore be used to upgrade the existing CollapsibleChannelMesh to a mesh that is derived from the Macro↩ElementNodeUpdateMesh base class. The dependence of the residuals of the "bulk" (fluid) elements on the geometric Data that affects its nodal positions is automatically taken into account if the "bulk" (fluid) elements (of type BULK_ELEMENT, say) are "upgraded" to the "wrapped" class MacroElementNodeUpdateElement<↩BULK_ELEMENT>. We refer to the discussion of the "toy" free-boundary Poisson problem for details of this methodology.

1.4.1.2 Representing the wall mesh as a GeomObject

What remains to be done is to represent the wall (discretised by the SolidFiniteElements contained in the wall mesh) as a GeomObject. For this purpose, oomph-lib provides the class

```
class MeshAsGeomObject : public GeomObject
```

whose constructor takes a pointer to the Mesh that is to be represented as a GeomObject. The Lagrangian and Eulerian dimensions of the mesh are taken to be the dimension of elements and the dimension of the nodes, respectively. The conversion from Mesh into GeomObject only makes sense if the element is derived from a FiniteElement, itself a GeomObject, which has associated Lagrangian (local) and Eulerian coordinate systems. In our collapsible channel problem, the Lagrangian dimension is one because the wall is parametrised by a single intrinsic coordinate (the shape of the mesh's constituent elements is parametrised by a single local coordinate), and the Eulerian dimension is two because the nodes in the wall mesh have two Eulerian coordinates. Assuming that Wall_mesh_pt stores a pointer to the wall mesh, the code

```
// Build a geometric object (one Lagrangian, two Eulerian coordinates)
// from the wall mesh
MeshAsGeomObject* wall_geom_object_pt=
 new MeshAsGeomObject(Wall_mesh_pt);
```

creates a GeomObject representation of the wall mesh that allows us to obtain the position of a material point on the wall (parametrised by its Lagrangian coordinate ξ) from the GeomObject::position(...) function. For instance, the following code computes the position vector $\mathbf{r}$ to the material point on the deformed wall, located at $\xi = 0.5$.

```
Vector<double> xi(1);
xi[0]=0.5;
Vector<double> r(2);
wall_geom_object->position(xi,r);
```

Here is a graphical illustration of the various representations of the domain boundary:

1. A "normal" GeomObject

In single-physics problems with prescribed boundary motion the domain boundary may be represented by a GeomObject. The GeomObject provides a parametrisation of its shape in terms of an intrinsic coordinate, ζ , as shown in this sketch:

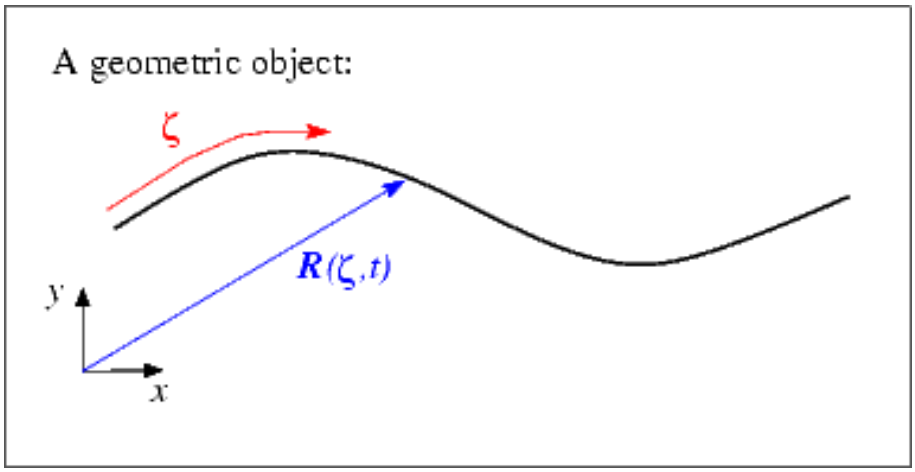

Figure 1.4 A geometric object parametrised by an intrinsic coordinate.

The function `GeomObject::position(zeta,r)` computes the position vector `r` to a point on the `GeomObject`, as identified by its intrinsic coordinate `zeta`.

2. A beam/shell structure

In beam/shell problems, the shape of the deformed structure is parametrised by its Lagrangian coordinate , ξ.

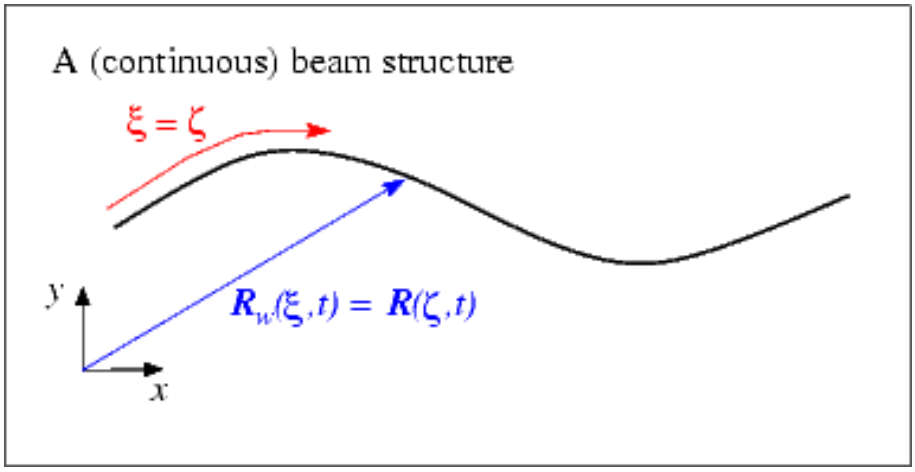

Figure 1.5 A (continuous) beam structure, parametrised by a Lagrangian coordinate.

Beam/shell structures may therefore act as `GeomObjects` if we interpret their Lagrangian coordinate, ξ , as the `GeomObject's` intrinsic coordinate, ζ.

3. A discretised beam/shell structure

In an `oomph-lib` computation, the beam/shell structure will, of course, have been discretised by a number of `SolidFiniteElements`. The `MeshAsGeomObject` discussed above, is therefore a "compound" `GeomObject` that contains a number of sub-objects – the mesh's constituent `SolidFiniteElements`. Within the "compound" `GeomObject` each sub-object acts as a `GeomObject` in its own right – the shape of a `SolidFiniteElement` is parametrised by its local coordinate `s`.

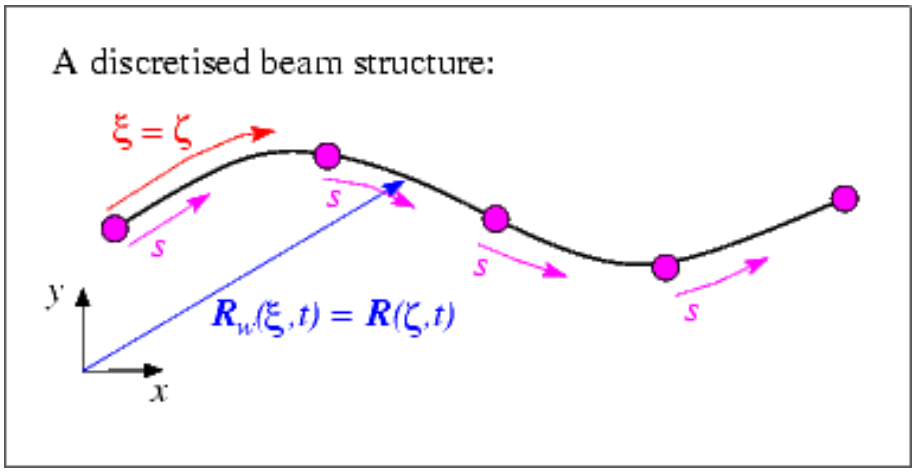

Figure 1.6 A discretised beam structure.

The `MeshAsGeomObject::position(...)` function therefore determines the position vector to the point labelled by the Lagrangian coordinate ξ (i.e. the `GeomObject's` intrinsic coordinate, ζ) in a two-stage process: First it determines which of its constituent `SolidFiniteElements` "contains" the relevant material point (This is possible because the function `SolidFiniteElement::interpolated_xi(...)` provides access to the Lagrangian coordinate inside the element) and then uses the `SolidFiniteElement::interpolated_x(...)` function to determine the Eulerian coordinates of that point.

1.4.2 Applying the fluid-traction to the wall elements: FSIWallElements

Next, we shall discuss how the fluid traction is added to the load terms in the wall equations. As mentioned above, the computation of the residuals of the wall equations requires the evaluation of the combined load vector at the Gauss points in the wall elements. Furthermore, the dependence of the residuals on those (fluid) degrees of freedom that affect the traction must be taken into account when computing the wall element's Jacobian matrix. Storage for the various lookup schemes required for such computations is provided in the virtual base class `FSIWallElement` whose inheritance structure is as follows:

```
class FSIWallElement : public virtual SolidFiniteElement,
                       public virtual ElementWithExternalElement
```

The `FiniteElement` class inherits from `GeomObject` and by default the `GeomObject::position(...)` function calls the function `FiniteElement::interpolate_x(..)`; in other words, the element's local coordinate is regarded as the intrinsic coordinate that parametrises its shape. Thus we already have a standard interface through which an `FSIWallElement` can be used to parametrise the shape of (part of) the domain boundary.
By inheriting from the `SolidFiniteElement` class, we establish that the shape of the `FSIWallElement` is determined by the positional `Data` stored at its constituent `SolidNodes`. Recall that this information is required during the computation of the shape derivatives of the fluid equations.
The `ElementWithExternalElement` class provides the generic storage and helper functions required to keep track of the external elements that are adjacent to Gauss points in any `FiniteElement`, see also the tutorial on `Boussinesq convection with a multi-domain approach.` In the `FSIWallElement`, it is the fluid elements that load the structure which are adjacent to the Gauss points. The various lookup schemes required to determine these fluid elements may be generated completely automatically by the helper function

```
template<class FLUID_ELEMENT, unsigned DIM_FLUID>
void FSI_functions::setup_fluid_load_info_for_solid_elements(
     Problem* problem_pt,
     const unsigned &boundary_in_fluid_mesh,
     Mesh* const& fluid_mesh_pt,
     Mesh* const& solid_mesh_pt);
```

which is defined in the namespace `FSI_functions`. The template parameters `FLUID_ELEMENT` and `DIM_FLUID` specify the type of the fluid element and its spatial (Eulerian) dimension, respectively. The arguments are the pointer to the problem, the number of the boundary in the fluid mesh adjacent to the elastic wall, and the pointers to the fluid and wall meshes. The function assumes that boundary coordinates have been set for the fluid nodes on the mesh boundary specified by the argument `boundary_in_fluid_mesh`, and that these boundary coordinates are consistent with the parametrisation of the wall mesh by its Lagrangian coordinate. (See `the discussion of the mesh generation procedures for domains with curvilinear boundaries` for a more detailed discussion of boundary coordinates for nodes.)
The `FSIWallElement` provides the protected member function `FSIWallElement::fluid_load_`

vector(...) which may be used in a specific FSIWallElement (such as the FSIHermiteBeam↩ Element) to compute the fluid traction (on the solid mechanics stress scale), and to add it to any external load that may already be acting on the element. The conversion from the fluid to the solid non-dimensionalisation of the traction is performed automatically by multiplying the traction vector (on the fluid stress scale) obtained from the "adjacent" FSIFluidElement by the stress ratio Q which has a default value of $Q = 1$. This default assignment may be overwritten by setting a pointer to a variable that specifies Q by using the function FSIWallElement↩ ::q_pt().
The class also overloads the SolidFiniteElement::fill_in_contribution_to_jacobian(...) function so that the derivatives of the residuals with respect to those unknowns that affect the fluid traction on the wall are included when computing the element's Jacobian matrix.
In the driver code, discussed below, we will discretise the wall with FSHermiteBeamElements, a class that is composed (by multiple inheritance) from the single-physics HermiteBeamElement and the FSIWall↩ Element base class:

```
class FSIHermiteBeamElement : public virtual HermiteBeamElement,
                              public virtual FSIWallElement
```

1.4.2.1 Obtaining the fluid traction from "adjacent" fluid elements: FSIFluidElements

Having provided a function that allows us to determine which fluid elements are located next to a given FSIWall↩ Element, we have to define standard interfaces through which we can obtain the traction that the "adjacent" fluid element exerts onto the wall. Furthermore, we have to determine which unknowns affect the fluid traction to enable us to evaluate the derivatives of the wall residuals with respect to these unknowns. Interfaces for the relevant functions are provided in the base class FSIFluidElement, whose most important member function (for the purpose of the present discussion) is FSIFluidElement::get_load(...). The purpose of this function is to compute the traction exerted by the FSIFluidElement onto the adjacent FSIWallElement, given the outer unit vector onto the FSIFluidElement. The FSIFluidElement class has two further pure virtual member functions whose role is to determine the unknowns (e.g. velocity and pressure values) that affect the fluid traction. For newly developed fluid elements these functions have to implemented on a case-by-case basis. However, all existing fluid elements in oomph-lib are already derived from the FSIFluidElement class and therefore provide a suitable implementation of these functions. It is therefore not necessary to explicitly "upgrade" oomph-lib's fluid elements before using them in FSI computations.

1.5 Overview of the driver code

The driver code for the fully-coupled fluid-structure interaction problem is a straightforward combination of the two single-physics codes with a few trivial additional steps. The main steps in the problem setup are:

1. Create the wall mesh, using FSIHermiteBeamElements instead of HermiteBeamElements.
2. **NEW:** Create a GeomObject representation of the wall mesh, using the MeshAsGeomObject class.
3. Create the fluid mesh, using elements of type MacroElementNodeUpdateElement<QCrouzeix↩ RaviartElement<2>> instead of QCrouzeixRaviartElement<2>. Use the MeshAsGeom↩ Object representation of the elastic wall, created in the previous step to represent the curvilinear domain boundary.
4. Build a mesh of traction elements that apply the prescribed-traction boundary condition at the inflow. Add all three meshes (fluid, solid and traction mesh) to the Problem's collection of sub-meshes and build the global mesh.
5. Pass the relevant function pointers (for Reynolds number, external loads, etc) to the various elements and apply the boundary conditions.
6. **NEW:** Call the function FSI_functions::setup_fluid_load_info_for_solid_elements(...) to set up the lookup scheme that establishes which fluid elements affect the traction on the wall.
7. Set up the equation numbering scheme.
8. Done! .

Most steps in this sequence are either identical to those in the corresponding single-physics codes, or require only trivial modifications. Each of the two new steps (2 and 6) can be implemented with a single line of code.

Consequently, most of the driver code, discussed in detail below, contains verbatim copies of code segments from the respective single-physics codes.

1.6 Namespace for the global physical variables

The namespace for the "global" physical parameters contains the same fluid parameters as in the collapsible channel problem with prescribed wall motion: We define the Reynolds and Womersley numbers and the fluid pressure at the upstream end, and provide a function that specifies the applied traction at the inflow:

```
//====start_of_physical_parameters====================
/// Namespace for phyical parameters
//====================================================
namespace Global_Physical_Variables
{
 /// Reynolds number
 double Re=50.0;

 /// Womersley = Reynolds times Strouhal
 double ReSt=50.0;

 /// Default pressure on the left boundary
 double P_up=0.0;

 /// Traction applied on the fluid at the left (inflow) boundary
 void prescribed_traction(const double& t,
                          const Vector<double>& x,
                          const Vector<double>& n,
                          Vector<double>& traction)
 {
  traction.resize(2);
  traction[0]=P_up;
  traction[1]=0.0;
 } //end traction
```

Next we define the wall parameters (wall thickness, prestress and external pressure) and assign default values. Note that the function that specifies the load on the wall contains only the load due to the external pressure – the additional load due to the fluid traction will be added automatically by the `FSIWallElements`, using the lookup scheme set up by the function `FSI_functions::setup_fluid_load_info_for_solid_elements(...)` discussed earlier.

```
 /// Non-dimensional wall thickness. As in Jensen & Heil (2003) paper.
 double H=1.0e-2;

 /// 2nd Piola Kirchhoff pre-stress. As in Jensen & Heil (2003) paper.
 double Sigma0=1.0e3;

 /// External pressure
 double P_ext=0.0;

 /// Load function: Apply a constant external pressure to the wall.
 /// Note:  This is the load without the fluid contribution!
 /// Fluid load gets added on by FSIWallElement.
 void load(const Vector<double>& xi, const Vector<double>& x,
           const Vector<double>& N, Vector<double>& load)
 {
  for(unsigned i=0;i<2;i++)
   {
    load[i] = -P_ext*N[i];
   }
 } //end of load
```

Finally, we define the interaction parameter Q and give it a default value.

```
 /// Fluid structure interaction parameter: Ratio of stresses used for
 /// non-dimensionalisation of fluid to solid stresses.
 double Q=1.0e-5;
} // end of namespace
```

1.7 The undeformed wall

We represent the undeformed geometry of the elastic wall, defined by equation (5), as a `GeomObject`, specifying the x-coordinate of its left end and its (constant) y-coordinate as arguments to the constructor:

```
//====start_of_underformed_wall=======================================
/// Undeformed wall is a steady, straight 1D line in 2D space
///  \f[ x = X_0 + \zeta \f]
///  \f[ y = H \f]
//====================================================================
class UndeformedWall : public GeomObject
{
```

```
public:

 /// Constructor: arguments are the starting point and the height
 /// above y=0.
 UndeformedWall(const double& x0, const double& h): GeomObject(1,2)
  {
   X0=x0;
   H=h;
  }
```

The two versions of the `position(...)` function are straightforward:

```
 /// Position vector at Lagrangian coordinate zeta
 void position(const Vector<double>& zeta, Vector<double>& r) const
  {
   // Position Vector
   r[0] = zeta[0]+X0;
   r[1] = H;
  }

 /// Parametrised position on object: r(zeta). Evaluated at
 /// previous timestep. t=0: current time; t>0: previous
 /// timestep. Calls steady version.
 void position(const unsigned& t, const Vector<double>& zeta,
               Vector<double>& r) const
  {
   // Use the steady version
   position(zeta,r);
  } // end of position
```

Since the `GeomObject` is used to specify the undeformed shape of a `HermiteBeamElement`, the function `GeomObject::d2position(...)` must be implemented to define the beam's curvature in the undeformed configuration.

```
 /// Posn vector and its  1st & 2nd derivatives
 /// w.r.t. to coordinates:
 /// \f$ \frac{dR_i}{d \zeta_\alpha}\f$ = drdzeta(alpha,i).
 /// \f$ \frac{d^2R_i}{d \zeta_\alpha d \zeta_\beta}\f$ =
 /// ddrdzeta(alpha,beta,i). Evaluated at current time.
 void d2position(const Vector<double>& zeta,
                 Vector<double>& r,
                 DenseMatrix<double> &drdzeta,
                 RankThreeTensor<double> &ddrdzeta) const
  {
   // Position vector
   r[0] = zeta[0]+X0;
   r[1] = H;
   // Tangent vector
   drdzeta(0,0)=1.0;
   drdzeta(0,1)=0.0;
   // Derivative of tangent vector
   ddrdzeta(0,0,0)=0.0;
   ddrdzeta(0,0,1)=0.0;
  } // end of d2position
```

The private member data contains the two geometric parameters.

```
 private :

 /// x position of the undeformed beam's left end.
 double X0;

 /// Height of the undeformed wall above y=0.
 double H;
}; //end_of_undeformed_wall
```

1.8 The driver code

As with most previous time-dependent codes, we use command line arguments to indicate if the code is run during `oomph-lib's` self-test procedures. If command line arguments are specified, we use a coarser discretisation and perform fewer timesteps. After storing the command line arguments, we choose the number of elements in the mesh, and set the lengths of the domain.

```
//============start_of_main============================================
/// Driver code for a collapsible channel problem with FSI.
/// Presence of command line arguments indicates validation run with
/// coarse resolution and small number of timesteps.
//======================================================================
int main(int argc, char* argv[])
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);

 // Reduction in resolution for validation run?
 unsigned coarsening_factor=1;
```

```
if (CommandLineArgs::Argc>1)
 {
  coarsening_factor=4;
 }
// Number of elements in the domain
unsigned nup=20/coarsening_factor;
unsigned ncollapsible=40/coarsening_factor;
unsigned ndown=40/coarsening_factor;
unsigned ny=16/coarsening_factor;
// Length of the domain
double lup=5.0;
double lcollapsible=10.0;
double ldown=10.0;
double ly=1.0;
```

We assign values for the external pressure (on the wall stiffness scale) and for the upstream fluid pressure (on the fluid pressure scale). The latter is again chosen so that in the absence of any wall deformation, the applied pressure difference would drive steady Poiseuille flow through the channel.

```
// Set external pressure (on the wall stiffness scale).
Global_Physical_Variables::P_ext = 1.0e-1;

// Pressure on the left boundary: This is consistent with steady
// Poiseuille flow
Global_Physical_Variables::P_up=12.0*(lup+lcollapsible+ldown);
```

We build the problem with 2D quadrilateral Crouzeix-Raviart elements, "upgraded" to `MacroElementNode↩ UpdateElements`. This ensures that the "shape derivatives" (the derivatives of the fluid residuals with respect to the solid mechanics degrees of freedom that affect the nodal positions in the fluid elements), are incorporated into the fluid elements' Jacobian matrices.

```
// Build the problem with QCrouzeixRaviartElements
FSICollapsibleChannelProblem
 <MacroElementNodeUpdateElement<QCrouzeixRaviartElement<2> > >
 problem(nup, ncollapsible, ndown, ny,
         lup, lcollapsible, ldown, ly);
```

We choose the timestepping parameters before assigning the initial conditions. (Preliminary computations showed that the system performs oscillations with approximately
unit period, so the chosen value for the timestep `dt` corresponds to a time-integration with about 40 timesteps per period.)

```
// Timestep. Note: Preliminary runs indicate that the period of
// the oscillation is about 1 so this gives us 40 steps per period.
double dt=1.0/40.0;
// Initial time for the simulation
double t_min=0.0;

// Maximum time for simulation
double t_max=3.5;
// Initialise timestep
problem.time_pt()->time()=t_min;
problem.initialise_dt(dt);
// Apply initial condition
problem.set_initial_condition();
```

Next we specify the output directory, open a trace file and document the initial conditions

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

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

// Output the initial solution
problem.doc_solution(doc_info, trace_file);
// Increment step number
doc_info.number()++;
```

The timestepping loop is identical to that in `the problem with prescribed wall motion:`

```
// Find number of timesteps (reduced for validation)
unsigned nstep = unsigned((t_max-t_min)/dt);
if (CommandLineArgs::Argc>1)
 {
  nstep=3;
 }

// Timestepping loop
for (unsigned istep=0;istep<nstep;istep++)
 {
  // Solve the problem
  problem.unsteady_newton_solve(dt);

  // Outpt the solution
  problem.doc_solution(doc_info, trace_file);
```

```
   // Step number
   doc_info.number()++;
  }
 // Close trace file.
 trace_file.close();
}//end of main
```

1.9 The problem class

The problem class is very similar to that used for the problem with prescribed wall motion. We specify the type of the fluid element as a template parameter and pass the number of elements and the lengths of the domain to the constructor:

```
//====start_of_problem_class=============================================
/// Problem class
//=======================================================================
template <class ELEMENT>
class FSICollapsibleChannelProblem : public Problem
{
 public :

/// Constructor: The arguments are the number of elements and
/// the lengths of the domain.
 FSICollapsibleChannelProblem(const unsigned& nup,
                       const unsigned& ncollapsible,
                       const unsigned& ndown,
                       const unsigned& ny,
                       const double& lup,
                       const double& lcollapsible,
                       const double& ldown,
                       const double& ly);

 /// Destructor (empty)
 ~FSICollapsibleChannelProblem(){}
```

We provide access functions to the (pointers to) the fluid mesh,

```
 /// Access function for the specific bulk (fluid) mesh
 MacroElementNodeUpdateCollapsibleChannelMesh<ELEMENT>* bulk_mesh_pt()
  {
   // Upcast from pointer to the Mesh base class to the specific
   // element type that we're using here.
   return dynamic_cast<
    MacroElementNodeUpdateCollapsibleChannelMesh<ELEMENT>*>
    (Bulk_mesh_pt);
  }
```

and the wall mesh:

```
 /// Access function for the wall mesh
 OneDLagrangianMesh<FSIHermiteBeamElement>* wall_mesh_pt()
  {
   return Wall_mesh_pt;
  } // end of access to wall mesh
```

Unlike the problem with prescribed wall motion, the FSI problem does not have any time-dependent boundary conditions, therefore the pure virtual functions `Problem::action_before_solve()` and `Problem::action_after_solve()` can remain empty, and the function `Problem::actions_before_implicit_timestep()` is not needed.

```
 /// Update the problem specs before solve (empty)
 void actions_before_newton_solve(){}

 /// Update the problem after solve (empty)
 void actions_after_newton_solve(){}
```

However, since the wall displacement (which is determined as part of the solution!) affects the nodal positions in the fluid mesh via the `MacroElement/Domain` - based node-update, the position of the fluid nodes must be updated whenever the Newton solver updates the unknowns. This is precisely what the function `Problem::actions_before_newton_convergence_check()` is for; see the discussion of oomph-lib's various "action" functions for more details.

```
 /// Update before checking Newton convergence: Update the
 /// nodal positions in the fluid mesh in response to possible
 /// changes in the wall shape
 void actions_before_newton_convergence_check()
  {
   Bulk_mesh_pt->node_update();
  }
```

The functions `doc_solution(...)` and `set_initial_condition()` do what they always do.

```
 /// Doc the solution
 void doc_solution(DocInfo& doc_info,ofstream& trace_file);
```

```
 /// Apply initial conditions
 void set_initial_condition();
```

The private member function `create_traction_elements(...)` is used to attach the applied traction elements to the upstream end of the channel, exactly as in the problem with prescribed wall motion.

```
private :

 /// Create the prescribed traction elements on boundary b
 void create_traction_elements(const unsigned &b,
                               Mesh* const &bulk_mesh_pt,
                               Mesh* const &traction_mesh_pt);
```

The private member data stores the problem parameters,

```
 /// Number of elements in the x direction in the upstream part of the channel
 unsigned Nup;

 /// Number of elements in the x direction in the collapsible part of
 /// the channel
 unsigned Ncollapsible;

 /// Number of elements in the x direction in the downstream part of the channel
 unsigned Ndown;

 /// Number of elements across the channel
 unsigned Ny;

 /// x-length in the upstream part of the channel
 double Lup;

 /// x-length in the collapsible part of the channel
 double Lcollapsible;

 /// x-length in the downstream part of the channel
 double Ldown;

 /// Transverse length
 double Ly;
```

the pointers to the fluid mesh,

```
 /// Pointer to the "bulk" mesh
 MacroElementNodeUpdateCollapsibleChannelMesh<ELEMENT>* Bulk_mesh_pt;
```

the surface mesh that contains the applied-traction elements,

```
 /// Pointer to the "surface" mesh that applies the traction at the
 /// inflow
 Mesh* Applied_fluid_traction_mesh_pt;
```

and the wall mesh,

```
 /// Pointer to the "wall" mesh
 OneDLagrangianMesh<FSIHermiteBeamElement>* Wall_mesh_pt;
```

as well as pointers to various control nodes

```
 /// Pointer to the left control node
 Node* Left_node_pt;

 /// Pointer to right control node
 Node* Right_node_pt;

 /// Pointer to control node on the wall
 Node* Wall_node_pt;
};//end of problem class
```

1.10 The problem constructor

We copy the various mesh parameters to the Problem's private data

```
//=====start_of_constructor======================================
/// Constructor for the collapsible channel problem
//================================================================
template <class ELEMENT>
FSICollapsibleChannelProblem<ELEMENT>::FSICollapsibleChannelProblem(
 const unsigned& nup,
 const unsigned& ncollapsible,
 const unsigned& ndown,
 const unsigned& ny,
 const double& lup,
 const double& lcollapsible,
 const double& ldown,
 const double& ly)
{
 // Store problem parameters
 Nup=nup;
 Ncollapsible=ncollapsible;
 Ndown=ndown;
 Ny=ny;
```

```
Lup=lup;
Lcollapsible=lcollapsible;
Ldown=ldown;
Ly=ly;
```

and increase the maximum value of the residual that is permitted during the Newton iteration to accommodate possible poor initial guesses for the solution.

```
// Overwrite maximum allowed residual to accomodate bad initial guesses
Problem::Max_residuals=1000.0;
```

We construct a `BDF<2>` - timestepper for the time-integration of the fluid equations and add it to the Problem's collection of timesteppers:

```
// Allocate the timestepper -- this constructs the Problem's
// time object with a sufficient amount of storage to store the
// previous timsteps.
add_time_stepper_pt(new BDF<2>);
```

The wall mesh is built as in the corresponding `single-physics beam problem:` We create the `GeomObject` that describes the undeformed wall shape and construct the wall mesh, this time with `FSIHermiteBeamElements:`

```
// Geometric object that represents the undeformed wall:
// A straight line at height y=ly; starting at x=lup.
UndeformedWall* undeformed_wall_pt=new UndeformedWall(lup,ly);
//Create the "wall" mesh with FSI Hermite elements
Wall_mesh_pt = new OneDLagrangianMesh<FSIHermiteBeamElement>
 //(2*Ncollapsible+5,Lcollapsible,undeformed_wall_pt);
 (Ncollapsible,Lcollapsible,undeformed_wall_pt);
```

We note that even though the same number of fluid and beam elements are used to discretise the common boundary, the discretisation of the two domains is **not** matching as the shape of the domain boundary is represented by piecewise cubic Hermite polynomials within the beam elements, and by piecewise quadratic Lagrange polynomials within the fluid elements. Mathematically, this does not cause any problems as both representations converge to the same boundary shape as the meshes are refined further and further. We stress that `oomph-lib` does not even require the number of elements along the interface to match; see Exercises .

The `MacroElement/Domain` - based fluid-mesh update requires the wall shape (which is now parametrised by the wall mesh's constituent elements) to be represented by single `GeomObject`. We create the required "compound" `GeomObject` using the `MeshAsGeomObject` class:

```
// Build a geometric object (one Lagrangian, two Eulerian coordinates)
// from the wall mesh
MeshAsGeomObject* wall_geom_object_pt=
 new MeshAsGeomObject(Wall_mesh_pt);
```

This "compound" `GeomObject` can now be used to build the fluid mesh, exactly as in `the corresponding single-physics fluids problem.` (The `MacroElementNodeUpdateCollapsibleChannelMesh` is a trivial extension of the `CollapsibleChannelMesh` from which it is derived; see the discussion of the `"toy" free-boundary Poisson problem` for details.)

```
//Build bulk (fluid) mesh
Bulk_mesh_pt =
 new MacroElementNodeUpdateCollapsibleChannelMesh<ELEMENT>
 (nup, ncollapsible, ndown, ny,
  lup, lcollapsible, ldown, ly,
  wall_geom_object_pt,
  time_stepper_pt());
```

As in `the fluids problem with prescribed wall motion` we use a "boundary-layer squash function" to distribute the fluid elements non-uniformly across the channel so that more elements are located inside the thin Stokes layers that are likely to develop near the wall.

```
// Set a non-trivial boundary-layer-squash function...
Bulk_mesh_pt->bl_squash_fct_pt() = &BL_Squash::squash_fct;
// ... and update the nodal positions accordingly
Bulk_mesh_pt->node_update();
```

We create the sub-mesh that stores the applied traction elements and attach the elements to the upstream end of the channel. The three sub-meshes are then combined into the `Problem's` single global mesh.

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

// Create prescribed-traction elements from all elements that are
// adjacent to boundary 5 (left boundary), but add them to a separate mesh.
create_traction_elements(5,Bulk_mesh_pt,Applied_fluid_traction_mesh_pt);

// Add the sub meshes to the problem
add_sub_mesh(Bulk_mesh_pt);
add_sub_mesh(Applied_fluid_traction_mesh_pt);
add_sub_mesh(Wall_mesh_pt);
// Combine all submeshes into a single Mesh
build_global_mesh();
```

We complete the build process for the fluid elements by passing the pointers to the relevant problem parameters to

the elements,

```
// Complete build of fluid mesh
//------------------------------

// Loop over the elements to set up element-specific
// things that cannot be handled by constructor
unsigned n_element=Bulk_mesh_pt->nelement();
for(unsigned e=0;e<n_element;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;

 } // end loop over elements
```

then we apply the boundary conditions for the fluid velocities:

- both axial and transverse velocities are pinned along the bottom and the top boundaries (boundaries 0, 2, 3 and 4)
- the transverse velocity is pinned along the in- and outflow (boundaries 1 and 5).

```
// Apply boundary conditions for fluid
//------------------------------------
//Pin the velocity on the boundaries
//x and y-velocities pinned along boundary 0 (bottom boundary) :
unsigned ibound=0;
unsigned num_nod= bulk_mesh_pt()->nboundary_node(ibound);
for (unsigned inod=0;inod<num_nod;inod++)
 {
  for(unsigned i=0;i<2;i++)
   {
    bulk_mesh_pt()->boundary_node_pt(ibound, inod)->pin(i);
   }
 }

//x and y-velocities pinned along boundaries 2, 3, 4 (top boundaries) :
for(ibound=2;ibound<5;ibound++)
 {
  num_nod= bulk_mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    for(unsigned i=0;i<2;i++)
     {
      bulk_mesh_pt()->boundary_node_pt(ibound, inod)->pin(i);
     }
   }
 }
//y-velocity pinned along boundary 1 (right boundary):
ibound=1;
num_nod= bulk_mesh_pt()->nboundary_node(ibound);
for (unsigned inod=0;inod<num_nod;inod++)
 {
  bulk_mesh_pt()->boundary_node_pt(ibound, inod)->pin(1);
 }
//y-velocity pinned along boundary 5 (left boundary):
ibound=5;
num_nod= bulk_mesh_pt()->nboundary_node(ibound);
for (unsigned inod=0;inod<num_nod;inod++)
 {
  bulk_mesh_pt()->boundary_node_pt(ibound, inod)->pin(1);
 }
//end of pin_velocity
```

The applied traction elements require a pointer to the prescribed traction function:

```
// Complete build of applied traction elements
//--------------------------------------------
// Loop over the traction elements to pass pointer to prescribed
// traction function
unsigned n_el=Applied_fluid_traction_mesh_pt->nelement();
for(unsigned e=0;e<n_el;e++)
 {
  // Upcast from GeneralisedElement to NavierStokes traction element
  NavierStokesTractionElement<ELEMENT> *el_pt =
   dynamic_cast< NavierStokesTractionElement<ELEMENT>*>(
    Applied_fluid_traction_mesh_pt->element_pt(e));

  // Set the pointer to the prescribed traction function
  el_pt->traction_fct_pt() = &Global_Physical_Variables::prescribed_traction;
 }
```

The wall elements require pointers to the various problem parameters and the pointer to the `GeomObject` that defines the beam's undeformed shape. Depending on the relative orientation of the fluid and solid meshes, the

normal vector used by the FSIWallElement to determine the traction exerted by the adjacent FSIFluid↩ Element may either point into or out of the fluid domain. By default, it is assumed that the normal points into the fluid – in this case the traction computed by the adjacent FSIFluidElement is added to the external load that acts on the FSIWallElement. Evaluating the direction of the normal (e.g by plotting the vector obtained from FSIHermiteBeamElement::get_normal(...)) in the present problem shows that the normal to the wall elements actually points out of the fluid domain, therefore the fluid traction acts in the opposite direction to that assumed in the original formulation. This may be rectified by setting the boolean flag FSIHermiteBeam↩ Element::normal_points_into_fluid() to false.

```
// Complete build of wall elements
//--------------------------------

//Loop over the elements to set physical parameters etc.
n_element = wall_mesh_pt()->nelement();
for(unsigned e=0;e<n_element;e++)
 {
  // Upcast to the specific element type
  FSIHermiteBeamElement *elem_pt =
   dynamic_cast<FSIHermiteBeamElement*>(wall_mesh_pt()->element_pt(e));

  // Set physical parameters for each element:
  elem_pt->sigma0_pt() = &Global_Physical_Variables::Sigma0;
  elem_pt->h_pt() = &Global_Physical_Variables::H;

  // Set the load vector for each element
  elem_pt->load_vector_fct_pt() = &Global_Physical_Variables::load;
  // Function that specifies the load ratios
  elem_pt->q_pt() = &Global_Physical_Variables::Q;
  // Set the undeformed shape for each element
  elem_pt->undeformed_beam_pt() = undeformed_wall_pt;
  // The normal on the wall elements as computed by the FSIHermiteElements
  // points away from the fluid rather than into the fluid (as assumed
  // by default)
  elem_pt->set_normal_pointing_out_of_fluid();
 } // end of loop over elements
```

Both ends of the beam are pinned:

```
// Boundary conditions for wall mesh
//----------------------------------
// Set the boundary conditions: Each end of the beam is fixed in space
// Loop over the boundaries (ends of the beam)
for(unsigned b=0;b<2;b++)
 {
  // Pin displacements in both x and y directions
  wall_mesh_pt()->boundary_node_pt(b,0)->pin_position(0);
  wall_mesh_pt()->boundary_node_pt(b,0)->pin_position(1);
 }
```

We choose two fluid control nodes in the middle of the inflow and outflow cross-sections to document the velocities, and choose a central node in the wall mesh to document its displacement.

```
//Choose control nodes
//---------------------

// Left boundary
ibound=5;
num_nod= bulk_mesh_pt()->nboundary_node(ibound);
unsigned control_nod=num_nod/2;
Left_node_pt= bulk_mesh_pt()->boundary_node_pt(ibound, control_nod);

// Right boundary
ibound=1;
num_nod= bulk_mesh_pt()->nboundary_node(ibound);
control_nod=num_nod/2;
Right_node_pt= bulk_mesh_pt()->boundary_node_pt(ibound, control_nod);


// Set the pointer to the control node on the wall
Wall_node_pt=wall_mesh_pt()->node_pt(Ncollapsible/2);
```

Finally, we set up the remaining fluid-structure interaction: The fluid nodes that are located on the moving wall remain attached to material particles on the wall. The no-slip condition (4) therefore implies that the fluid velocity at each of these nodes must be equal to the nodes' velocity. Hence the fluid velocity must be updated whenever a node update function changes the nodal position. This is done most easily by means of the auxiliary node update function – a function that is executed automatically whenever a node's node_update() function is called. To achieve this we pass a function pointer to the FSI_functions::apply_no_slip_on_moving_wall() function to the nodes on the fluid mesh's boundary 3:

```
// Setup FSI
//----------
// The velocity of the fluid nodes on the wall (fluid mesh boundary 3)
// is set by the wall motion -- hence the no-slip condition must be
// re-applied whenever a node update is performed for these nodes.
```

```
 // Such tasks may be performed automatically by the auxiliary node update
 // function specified by a function pointer:
 ibound=3;
 num_nod= bulk_mesh_pt()->nboundary_node(ibound);
 for (unsigned inod=0;inod<num_nod;inod++)
  {
   bulk_mesh_pt()->boundary_node_pt(ibound, inod)->
    set_auxiliary_node_update_fct_pt(
     FSI_functions::apply_no_slip_on_moving_wall);
  }
```

Next, the `FSIHermiteBeamElements` have to be "told" which fluid elements are located next to their Gauss points to allow them to work out the fluid traction. The required lookup tables are created by the function `FSI_functions::setup_fluid_load_info_for_solid_elements(...)`:

```
 // Work out which fluid dofs affect the residuals of the wall elements:
 // We pass the boundary between the fluid and solid meshes and
 // pointers to the meshes. The interaction boundary is boundary 3 of the
 // 2D fluid mesh.
 FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
  (this,3,Bulk_mesh_pt,Wall_mesh_pt);
```

Finally, we set up the equation numbering scheme.

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

}//end of constructor
```

1.11 Post processing

The function `doc_solution(...)` outputs the velocity and wall displacement fields and records the time-trace of the axial velocity at the control nodes and the position of the wall's midpoint. The function `FSI_functions::doc_fsi(...)` is a helper function that can be used to document/validate the various FSI lookup schemes; see Comments and Exercises for an illustration of their output.

```
//====start_of_doc_solution==============================================
/// Doc the solution
//========================================================================
template <class ELEMENT>
void FSICollapsibleChannelProblem<ELEMENT>:: doc_solution(DocInfo& doc_info,
                                                          ofstream& trace_file)
{
 // Doc fsi
   FSI_functions::doc_fsi<MacroElementNodeUpdateNode>(Bulk_mesh_pt,Wall_mesh_pt,doc_info);
 ofstream some_file;
 char filename[100];
 // Number of plot points
 unsigned npts;
 npts=5;
 // Output fluid solution
 sprintf(filename,"%s/soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 bulk_mesh_pt()->output(some_file,npts);
 some_file.close();
 // Document the wall shape
 sprintf(filename,"%s/beam%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 wall_mesh_pt()->output(some_file,npts);
 some_file.close();

 // Write trace file
 trace_file << time_pt()->time() << " "
            << Wall_node_pt->x(1) << " "
            << Left_node_pt->value(0) << " "
            << Right_node_pt->value(0) << " "
            << Global_Physical_Variables::P_ext  << " "
            << std::endl;
} // end_of_doc_solution
```

1.12 Creation of the traction elements

This function is the same as the one used in the problem with prescribed wall motion.

1.13 Applying the initial conditions

This function is the same as the one used in the problem with prescribed wall motion.

1.14 Comments and Exercises

1.14.1 Comments

- **Variables that affect the fluid traction are not just velocities and pressures!**

  In the section Applying the fluid-traction to the wall elements: FSIWallElements we briefly discussed how the function `FSI_functions::setup_fluid_load_info_for_solid_elements(...)` determines the variables that affect the fluid traction onto the `FSIWallElements`. For a Newtonian fluid, the components of the fluid traction vector (on the viscous scale) onto the wall is given by

  $$t_i = -pN_i + \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) N_j \quad \text{for } i = 1, 2[, 3]$$

  where N_i (for $i = 1, 2[, 3]$) are the components of the normal to the fluid domain, pointing into the fluid. This equation shows that the fluid traction is primarily affected by the velocity and pressure degrees of freedom in the fluid elements that are "adjacent" to a given `FSIWallElement`.

  However, since the traction involves derivatives of the velocity, the traction is also affected by changes to the
  geometry of the fluid element. Therefore, the list of variables that affect the fluid traction must (and indeed does) include all those `Data` objects that are involved in the adjacent fluid elements' node update operations.

  Here is an animation that illustrates these dependencies for a relatively coarse discretisation in which the collapsible section of the fluid mesh is discretised with 10 "vertical columns" of `QCrouzeixRaviart↩Element<2>` elements, while the wall is discretised with 22 `FSIHermiteBeamElements`, each of which contains 3 Gauss points. The animation shows the region of the fluid mesh close to the (strongly deformed) elastic wall. Each different frame illustrates the FSI lookup schemes for a different wall element.

  - The position of the wall Gauss points are displayed by "gradient" markers while the corresponding points in the adjacent fluid elements (i.e. the points at which the fluid traction is computed) are displayed by "delta" markers. Since the fluid and solid discretisations are non-matching the points do not coincide exactly though they will continue to approach each other under further mesh refinement.
  - The coloured numbers indicate the number of values that affect the fluid traction on this `FSIWall↩Element`:
    - The red numbers represent the number of nodal values at the nodes of the adjacent fluid element(s) that affect the traction: For a 2D Crouzeix-Raviart element, each fluid node stores two velocity degrees of freedom, both of which affect the traction.
    - The blue numbers represent the number of internal `Data` values stored in an adjacent fluid element that affect the traction: In a 2D Crouzeix-Raviart element, each element stores three pressure values in its internal `Data` and all three pressure values affect the traction.
    - Finally, the green numbers indicate the number of `Data` values that are (potentially) involved in the node-update operation for the nodes in the adjacent fluid elements. Recall that during the `MacroElement` - based node-update we only refer to the wall via its representation as a `Wall↩AsGeomObject`, i.e. as a "compound" `GeomObject`. The geometric `Data` of a compound `GeomObject` is given by the geometric `Data` of *all* its sub-objects. Therefore, the geometric `Data` of the `WallAsGeomObject` includes the positional `Data` of all `SolidNodes` stored in this mesh. In an `FSIHermiteBeamElement`, each `SolidNode` stores four values, representing the node's x- and y-positions and their derivatives with respect to the element's local coordinate.

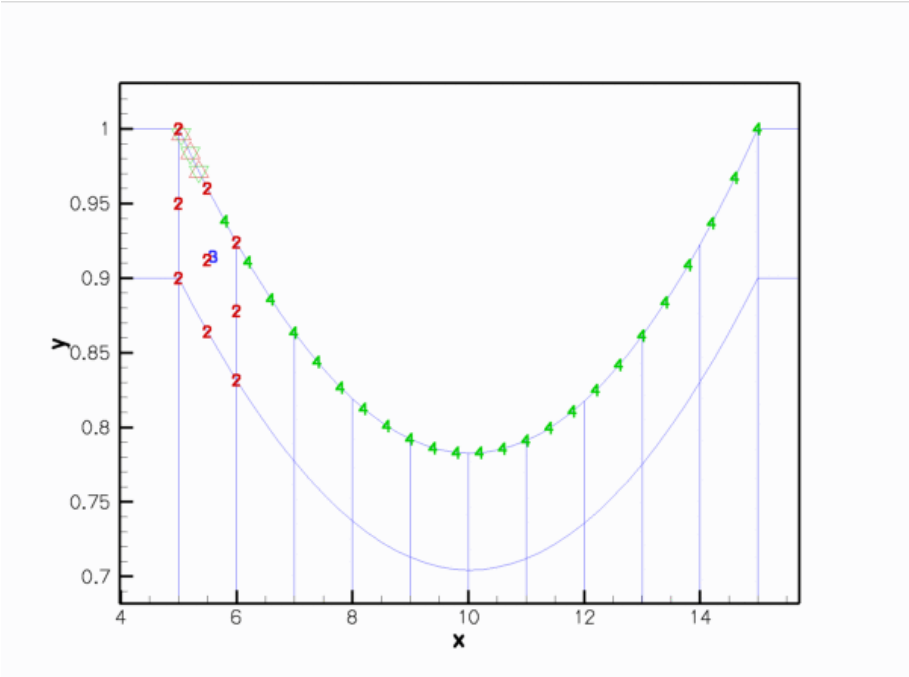

Figure 1.7 Animation of the Data values that affect the fluid traction that the adjacent fluid elements exert onto the various FSIHermiteBeamElements in the wall mesh. (The fluid elements are 2D Crouzeix-Raviart elements.)

Here is the corresponding animation for a discretisation with 2D Taylor-Hood elements. These elements have no internal `Data` but the pressure degrees of freedom are stored at the fluid element's corner nodes:

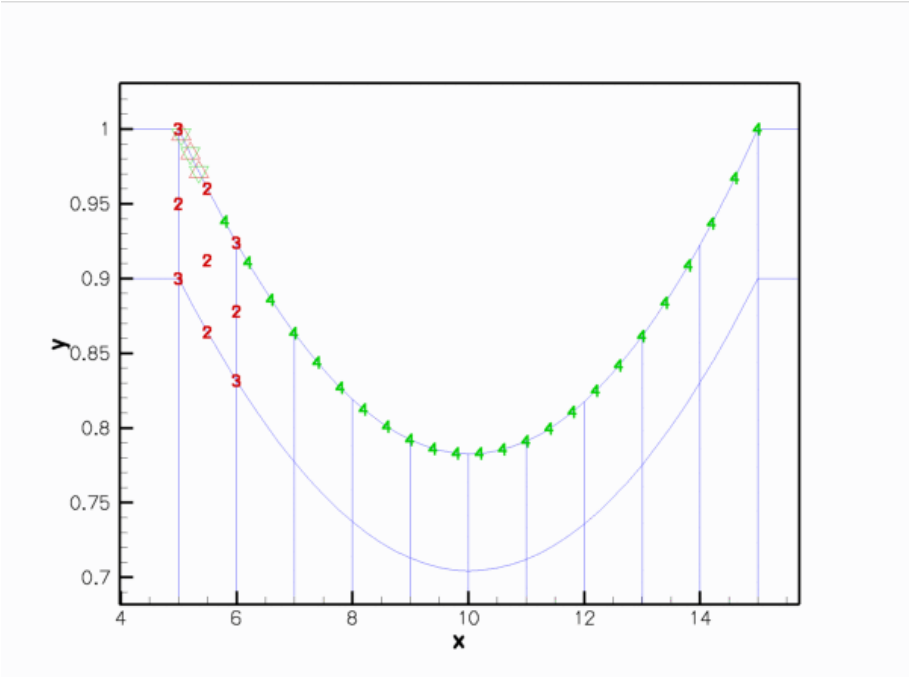

Figure 1.8 Animation of the Data values that affect the fluid traction that the adjacent fluid elements exert onto the various FSIHermiteBeamElements in the wall mesh. (The fluid elements are 2D Taylor-Hood elements.)

Finally, here is an animation that shows the (wall) degrees of freedom that affect the node-update of a given fluid node. The red square marker shows the fluid node; the green numbers show the number of the degrees of freedom at the `SolidNodes` that are involved that fluid node's node update. Again it is clear that the `MacroElement/`↩

Domain-based node update procedure in which the wall mesh is represented by a compound GeomObject does not result in a sparse node-update procedure: Each SolidNode in the wall mesh is assumed to affect the position of all fluid nodes.

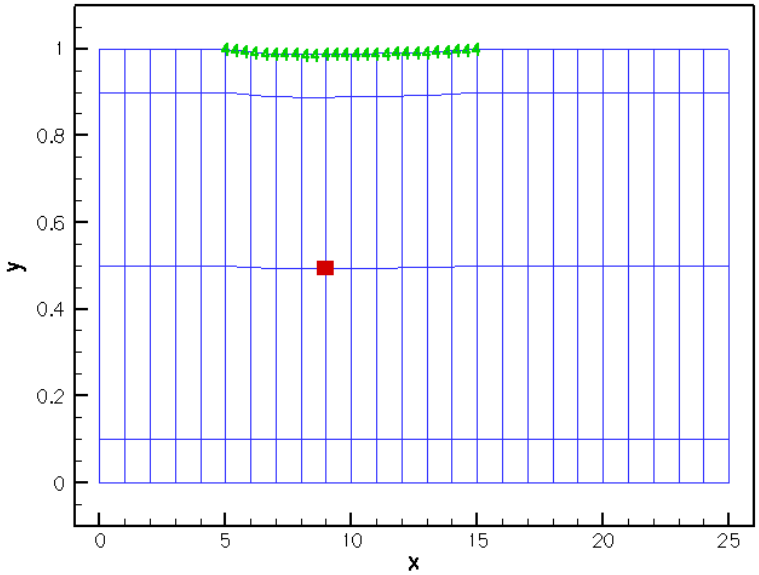

Figure 1.9 Animation of the Data values that affect the node update of the fluid nodes.

- **(In-)efficiency of the MacroElement-based node-update**

  The animations shown above illustrate very graphically that the implementation of the fluid-structure interaction via MacroElement/Domain - based node updates does not lead to a particularly efficient algorithm. The current approach suffers from two main problems:

  1. The fluid-node update is not sparse: Since we cannot distinguish between the various sub-objects in the "compound" GeomObject, we can do no better than assuming the worst-case scenario, namely that *all* positional degrees of freedom of *all* SolidNodes in the wall mesh potentially affect the nodal position in the fluid elements adjacent to the wall. Consequently, each FSIHermiteBeamElement in the wall mesh depends on all solid mechanics degrees of freedom in the wall mesh. As a result, the wall discretisation completely loses its sparsity!
  2. When updating the position of the fluid nodes via the fluid element's Domain/MacroElement representation, we obtain the wall shape from the GeomObject::position(...) function of the "compound" GeomObject, using the wall's Lagrangian coordinate ξ as the "compound" GeomObject's intrinsic coordinate. As discussed in the section Representing the wall mesh as a GeomObject, this is a very costly operation, since we first have to determine which of the constituent FSIHermiteBeam↩Elements "contains" the required Lagrangian coordinate, and then evaluate the Eulerian position of the relevant point in the element.

  In the next example we will demonstrate how the use of the algebraic node update procedure, described in an earlier example, allows us to avoid both problems, resulting in a much more efficient code.

1.14.2 Exercises

1. Use the function FSIHermiteBeamElement::get_normal(...) to plot the unit normal vector to the wall and thus confirm that the value for FSIHermiteBeamElement::normal_points_into_↩fluid() is correct. [**Hint:** Another way to sanity-check that the correct value for this flag has been set is to change the velocity boundary conditions at the upstream end to a pure Dirichlet condition by prescribing the

axial velocity profile. With Dirichlet conditions everywhere, one fluid pressure degree of freedom, p_{fix} , say, can (indeed must!) then be assigned arbitrarily. Now set the inflow velocity to zero and increase the value of p_{fix} from zero, say. If the wall collapses inwards as p_{fix} is increased the direction of normal was chosen wrongly!]

2. In section Applying the fluid-traction to the wall elements: FSIWallElements we mentioned that the function `FSI_functions::setup_fluid_load_info_for_solid_elements(...)` assumes that the fluid nodes on the FSI boundary store the boundary coordinate. Investigate what happens if this step is omitted, e.g. by commenting out the assignment of boundary coordinates with `Node::set_coordinates_on_boundary(...)` in `collapsible_channel_mesh.template.cc`.

3. In section The undeformed wall we mentioned that the function `GeomObject::d2position(...)` must be implemented for all `GeomObjects` that specify the undeformed shape of a beam element. Check what happens if this function is not implemented, e.g. by commenting out its definition in the `UndeformedWall` class.

4. In section The problem constructor we commented that `oomph-lib` does not require the discretisations of the fluid and solid meshes to match along the common boundary. Confirm this, e.g., by increasing the number of elements in the wall mesh. .

1.15 Source files for this tutorial

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

  `demo_drivers/interaction/fsi_collapsible_channel/`

- The driver code is:

  `demo_drivers/interaction/fsi_collapsible_channel/fsi_collapsible_channel.cc`

1.16 PDF file

A `pdf version` of this document is available.