Chapter 1

Demo problem: Flow in a 2D channel with an elastic leaflet

In this example we consider the flow in a 2D channel which is partially obstructed by an elastic leaflet – the FSI generalisation of the Navier-Stokes problem in which the motion of the leaflet is prescribed. A particular feature of this problem is that the leaflet (modelled as a thin-walled beam structure) is totally immersed in the fluid and is therefore exposed to the fluid traction from both sides.

The problem presented here was used as one of the test cases for oomph-lib's FSI preconditioner; see

Heil, M., Hazel, A.L. & Boyle, J. (2008): Solvers for large-displacement fluid-structure interaction problems: Segregated vs. monolithic approaches. Computational Mechanics.

In this tutorial we concentrate on the problem formulation. The application of the preconditioner is discussed elsewhere – the required source code is contained in the driver code.

1.1 The Problem

The figure below shows a sketch of the problem: A 2D channel of height H^*_{tot} and length $L^*_{left} + L^*_{right}$ is partially occluded by a thin-walled elastic leaflet of height $H^*_{leaflet}$. (As usual, we use asterisks to distinguish dimensional quantities from their non-dimensional equivalents to be introduced later.) We assume that the leaflet is sufficiently thin so that, as far as the boundary conditions for the fluid are concerned, the leaflet can be assumed to be infinitely thin. This allows us to parametrise its shape by a single Lagrangian coordinate ξ^*. Hence we write the position vector to a material point on the leaflet as $\mathbf{R}^*_w(\xi^*, t^*)$. A pulsatile Poiseuille flow whose mean velocity fluctuates between U^* and $2U^*$ is imposed at the upstream end of the channel. The outflow is assumed to be parallel and axially traction-free.

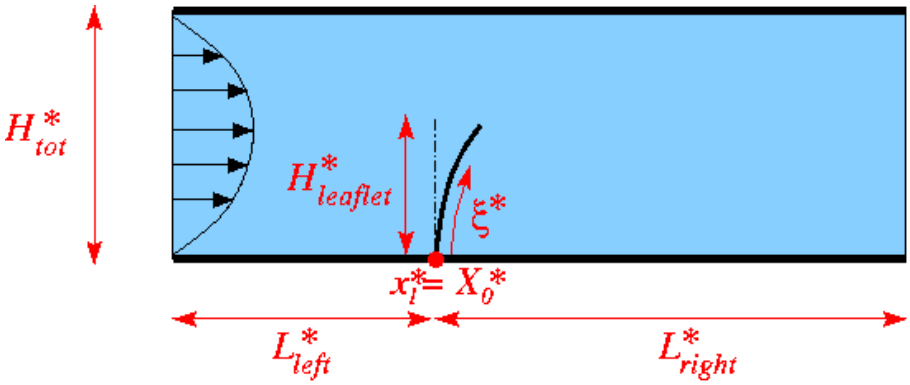

Figure 1.1 Sketch of the problem.

We non-dimensionalise all length and coordinates on the channel width, H^*_{tot} , time on the natural timescale of the flow, H^*_{tot}/U^*, the velocities on the (minimum) mean velocity,
U^*, and the pressure on the viscous scale.
The fluid flow is then governed by the non-dimensional Navier-Stokes equations

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

where $Re=\rho U^* H^*_0/\mu$ and $St=1$, and

$$\frac{\partial u_i}{\partial x_i}=0,$$

subject to a time-periodic, parabolic inflow with non-dimensional period T,

$$\mathbf{u}=6x_2(1-x_2)\left(1+\frac{1}{2}(1-\cos(2\pi t/T))\right)\mathbf{e}_1$$

in the inflow cross-section; parallel, axially-traction-free outflow at the outlet; and no-slip on the stationary channel walls, $\mathbf{u}=\mathbf{0}$. The no-slip condition on the leaflet is

$$\mathbf{u}=\frac{\partial \mathbf{R}_w(\xi,t)}{\partial t}.$$

We model the leaflet as a thin-walled, massless, elastic Kirchhoff-Love beam of wall thickness h^*. The beam's 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 by the principle of virtual displacements, discussed in detail in another tutorial. As in the Navier-Stokes equations, we scale all lengths in the beam problem on the channel's width, H^*_{tot}. The non-dimensional position vector $\mathbf{r}_w(\xi)$ to the undeformed wall is then given by

$$\mathbf{r}_w(\xi)=\mathbf{R}_w(\xi,t=0)=\begin{pmatrix}X_0\\ \xi\end{pmatrix}\qquad \text{where } \xi\in[0,H_{leaflet}].$$

Our non-dimensionalisation of the principle of virtual displacements requires all stresses and tractions to be non-dimensionalised on the beam's (effective 1D) elastic modulus, E_{eff}. The non-dimensional load vector $\mathbf{f}=\mathbf{f}^*/E_{eff}$ that acts on the leaflet (combining the fluid tractions acting on its front and back faces) is then given by

$$f_i=Q\Bigg\{\left(p\big|_{front}\,N_i^{[front]}-\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)\Big|_{front}N_j^{[front]}\right)+$$
$$+\left(p\big|_{back}\,N_i^{[back]}-\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)\Big|_{back}N_j^{[back]}\right)\Bigg\}\qquad \text{for } i=1,2,$$

.

where

$$Q=\frac{\mu U}{E_{eff}H^*_{tot}}$$

is the ratio of the fluid pressure scale, $\mu U/H^*_{tot}$, used to non-dimensionalise the Navier-Stokes equations, to the beam's effective elastic modulus, E_{eff}. The parameter Q therefore indicates the strength of the fluid-structure interaction. In particular, if $Q=0$ the leaflet does not "feel" the fluid traction. $\mathbf{N}_{front}$ and $\mathbf{N}_{back}$ are the outer unit normals on the "front" and "back" faces of the deformed leaflet, as shown in this sketch of the non-dimensional version of the problem:

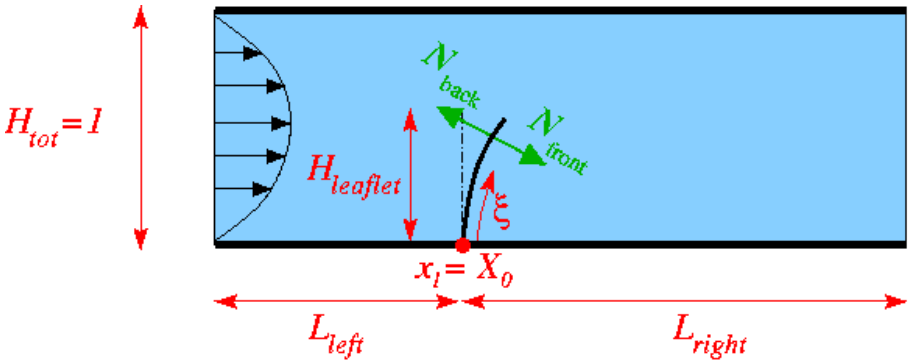

Figure 1.2 Sketch of the problem in dimensionless variables.

1.2 Results

The figure below shows (a) a snapshot of the flow field
(pressure contours and instantaneous streamlines) and (b) the time-trace of the horizontal position of the leaflet's tip for a Reynolds number of $Re = 200$, a non-dimensional period of $T = 2$, and an interaction parameter of $Q = 10^{-6}$. Following the decay of initial transients the leaflet performs periodic large-amplitude oscillations with the period of the pulsating inflow. Large velocity gradients develop at the front of the leaflet and in the shear layer that emanates from its tip and separates the recirculating flow region behind the leaflet from the main flow. Fig. (c) illustrates the non-uniform mesh refinement and shows the improved resolution in the high-shear regions, particularly near the leaflet's tip where the pressure is singular. The mesh was continuously adapted throughout the simulation and contained an average of about 32,000 degrees of freedom. This is a fraction of the 1,324,343 degrees of freedom that would be required to achieve the same local resolution via uniform mesh refinement.

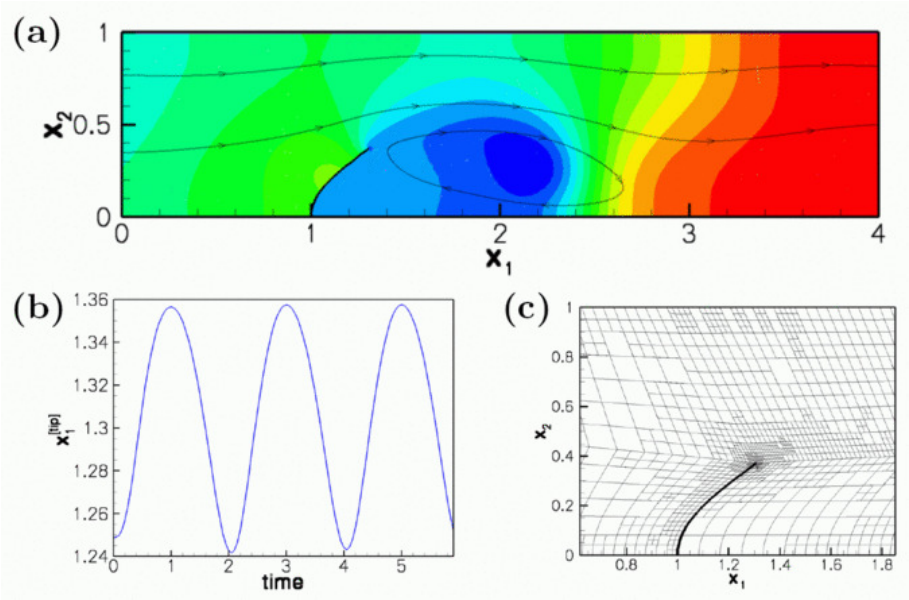

Figure 1.3 Computational results.

The corresponding animation illustrates the algebraic node update strategy (implemented with an AlgebraicMesh, discussed in more detail in another tutorial) and the evolution of the flow field. Note that the instantaneous streamlines intersect the (impermeable) leaflet because the leaflet is not stationary. The animation shows how the mesh adapts itself to changes in the flow field – using much smaller elements in high-shear regions, including the artificial outflow boundary layer that is created by the imposition of parallel outflow in a region where the flow is far from fully-developed.

1.3 The global parameters

As usual we use a namespace to define the problem parameters: The Reynolds number and its product with the Strouhal number (both initialised to 50),

```
//==== start_of_global_parameters=================================
/// Global parameters
//================================================================
namespace Global_Physical_Variables
{
 /// Reynolds number
 double Re=50.0;

 /// Womersley number: Product of Reynolds and Strouhal numbers
 double ReSt=50.0;
```

the leaflet's non-dimensional wall thickness, $h = h^*/H^*_{tot}$, and the FSI parameter,

```
 /// Non-dimensional wall thickness.
 double H=0.05;
```

```
 /// Fluid structure interaction parameter: Ratio of stresses used for
 /// non-dimensionalisation of fluid to solid stresses.
 double Q=1.0e-6;
```

and the parameters that define the magnitude of the pulsating inflow.

```
 /// Period for fluctuations in flux
 double Period=2.0;

 /// Min. flux
 double Min_flux=1.0;

 /// Max. flux
 double Max_flux=2.0;

 /// Flux: Pulsatile flow fluctuating between Min_flux and Max_flux
 /// with period Period
 double flux(const double& t)
 {
  return Min_flux+
    (Max_flux-Min_flux)*0.5*(1.0-cos(2.0*MathematicalConstants::Pi*t/Period));
 }
} // end_of_namespace
```

1.4 The undeformed leaflet

We use a `GeomObject` to describe the initial, stress-free shape of the elastic leaflet: a vertical straight line. The member function `d2position(...)` provides the first and second derivatives of the position vector, as required by the variational principle that governs the beam's deformation; see the beam theory tutorial for details.

```
//=====start_of_undeformed_leaflet================================
/// GeomObject: Undeformed straight, vertical leaflet
//================================================================
class UndeformedLeaflet : public GeomObject
{
public:

 /// Constructor: argument is the x-coordinate of the leaflet
 UndeformedLeaflet(const double& x0): GeomObject(1,2)
  {
   X0=x0;
  }

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

 /// 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

 /// 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] = X0;
   r[1] = zeta[0];
   // Tangent vector
   drdzeta(0,0)=0.0;
   drdzeta(0,1)=1.0;
   // Derivative of tangent vector
   ddrdzeta(0,0,0)=0.0;
   ddrdzeta(0,0,1)=0.0;
  } // end of d2position
```

```
 /// Number of geometric Data in GeomObject: None.
 unsigned ngeom_data() const {return 0;}
 private :

 /// x position of the undeformed leaflet's origin.
 double X0;
}; //end_of_undeformed_wall
```

1.5 The driver code

As with most time-dependent demo codes, the specification of a non-zero number of command line arguments will be interpreted as the code being run as part of `oomph-lib's` self-test in which case only a few timesteps will be performed. Therefore we start by storing the command line arguments in the namespace `CommandLineArgs` to make them accessible throughout the code.

```
//======= start_of_main==============================================
/// Driver code  -- pass a command line argument if you want to run
/// the code in validation mode where it only performs a few steps
//====================================================================
int main(int argc, char* argv[])
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);
```

Next we specify the geometry and mesh parameters, and build the `Problem` object, using the `Algebraic↩ Element` version of the two-dimensional `RefineableQTaylorHoodElements`.

```
 //Parameters for the leaflet: x-position of root and height
 double x_0 = 1.0;
 double hleaflet=0.5;
 // Number of elements in various regions of mesh
 unsigned nleft=6;
 unsigned nright=18;
 unsigned ny1=3;
 unsigned ny2=3;
 // Dimensions of fluid mesh: length to the left and right of leaflet
 // and total height
 double lleft =1.0;
 double lright=3.0;
 double htot=1.0;

 //Build the problem
 FSIChannelWithLeafletProblem<
  AlgebraicElement<RefineableQTaylorHoodElement<2> > >
  problem(lleft,lright,hleaflet,
          htot,nleft,nright,ny1,ny2,x_0);
```

We prepare a `DocInfo` object and open a trace file to document the system's evolution.

```
 // Set up doc info
 DocInfo doc_info;
 doc_info.set_directory("RESLT");
 // Trace file
 ofstream trace;
 char filename[100];
 sprintf(filename,"%s/trace.dat",doc_info.directory().c_str());
 trace.open(filename);
```

Next we assign the timestepping parameters (using fewer timesteps if the code is run during a self-test) and document the system's initial state which provides the initial guess for the Newton iteration in the subsequent steady solve.

```
 // Number of timesteps (reduced for validation)
 unsigned nstep=200;
 if (CommandLineArgs::Argc>1)
  {
   nstep=2;
  }
 //Timestep:
 double dt=0.05;

 // Initialise timestep
 problem.initialise_dt(dt);
 // Doc initial guess for steady solve
 problem.doc_solution(doc_info,trace);
 doc_info.number()++;
```

We wish to start the time-dependent simulation from an initial condition that corresponds to the steady solution of the problem with unit influx (i.e. the steady solution obtained if the time-dependent boundary condition that is applied at the inlet is held fixed at its value for $t = 0$). For the required Reynolds number of $Re = 200$, the currently assigned initial guess (zero velocity and pressure, with the wall
in its undeformed position) is "too far" from the actual solution and the Newton method would diverge (try it!). Therefore we generate the initial steady solution in a preliminary sequence of steady computations in which we first compute the solution for a Reynolds number of $Re = 50$ (for this value the Newton method does converge).

This solution is then used as the initial guess for the computation at $Re = 100$, etc. We allow for up to 3 mesh adaptations for each Reynolds number to allow the mesh to adapt itself to the flow conditions before starting the time-dependent run.

```
// Initial loop to increment the Reynolds number in sequence of steady solves
//-------------------------------------------------------------------------
unsigned n_increment=4;
// Just to one step for validation run
if (CommandLineArgs::Argc>1)
 {
  n_increment=1;
 }
// Set max. number of adaptations
unsigned max_adapt=3;
Global_Physical_Variables::Re=0.0;
for (unsigned i=0;i<n_increment;i++)
 {
  // Increase Re and ReSt (for St=1)
  Global_Physical_Variables::Re+=50.0;
  Global_Physical_Variables::ReSt=Global_Physical_Variables::Re;
  // Solve the steady problem
  std::cout << "Computing a steady solution for Re="
            <<  Global_Physical_Variables::Re << std::endl;
  problem.steady_newton_solve(max_adapt);
  problem.doc_solution(doc_info,trace);
  doc_info.number()++;
 } // reached final Reynolds number
```

Finally, we start the proper timestepping procedure, allowing for one mesh adaptation per timestep and suppressing the re-assignment of the initial condition after the mesh adaptation by setting the `first` flag to false.

```
// Proper time-dependent run
//--------------------------
// Limit the number of adaptations during unsteady run to one per timestep
max_adapt=1;

// Don't re-set the initial conditions when adapting the mesh
bool first = false;
// Timestepping loop
for (unsigned istep=0;istep<nstep;istep++)
 {
  // Solve the problem
  problem.unsteady_newton_solve(dt,max_adapt,first);

  // Output the solution
  problem.doc_solution(doc_info,trace);

  // Step number
  doc_info.number()++;
 }
}//end of main
```

1.6 The Problem class

The `Problem` class contains the usual member functions, most of which are either empty or explained in more detail below.

```
//=====start_of_problem_class=============================================
/// FSI leaflet in channel
//========================================================================
template<class ELEMENT>
class FSIChannelWithLeafletProblem : public Problem
{
public:

 /// Constructor: Pass the lenght of the domain at the left
 /// of the leaflet lleft,the lenght of the domain at the right of the
 /// leaflet lright,the height of the leaflet hleaflet, the total height
 /// of the domain htot, the number of macro-elements at the left of the
 /// leaflet nleft, the number of macro-elements at the right of the
 /// leaflet nright, the number of macro-elements under hleaflet ny1,
 /// the number of macro-elements above hleaflet ny2, the abscissa
 /// of the origin of the leaflet x_0.
 FSIChannelWithLeafletProblem(const double& lleft,
                              const double& lright, const double& hleaflet,
                              const double& htot,
                              const unsigned& nleft, const unsigned& nright,
                              const unsigned& ny1, const unsigned&  ny2,
                              const double& x_0);

 /// Destructor empty
 ~FSIChannelWithLeafletProblem(){}

 /// Actions after solve (empty)
 void actions_after_newton_solve(){}
```

```
 /// Actions before solve (empty)
 void actions_before_newton_solve(){}

 /// Actions after adaptation
 void actions_after_adapt();

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

 /// Access function to fluid mesh
 RefineableAlgebraicChannelWithLeafletMesh<ELEMENT>* fluid_mesh_pt()
  {
   return Fluid_mesh_pt;
  }

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

Two member functions are implemented here: The function `actions_before_implicit_timestep()` updates the time-dependent velocity profile at the upstream boundary (boundary 3; see the enumeration of the mesh boundaries in the `ChannelWithLeafletMesh`):

```
/// Update the inflow velocity
 void actions_before_implicit_timestep()
  {
   // Actual time
   double t=time_pt()->time();
   // Amplitude of flow
   double ampl=Global_Physical_Variables::flux(t);
   // Update parabolic flow along boundary 3
   unsigned ibound=3;
   unsigned num_nod= Fluid_mesh_pt->nboundary_node(ibound);
   for (unsigned inod=0;inod<num_nod;inod++)
   {
    double ycoord = Fluid_mesh_pt->boundary_node_pt(ibound,inod)->x(1);
    double uy = ampl*6.0*ycoord/Htot*(1.0-ycoord/Htot);
    Fluid_mesh_pt->boundary_node_pt(ibound,inod)->set_value(0,uy);
    Fluid_mesh_pt->boundary_node_pt(ibound,inod)->set_value(1,0.0);
   }
  } // end of actions_before_implicit_timestep
```

Since the node-update is performed by an algebraic node-update procedure the nodal positions in the fluid mesh must be updated whenever any of the (solid) degrees-of-freedom change. This is done automatically during the computation of the shape derivatives (implemented in the `AlgebraicElement` wrapper class described `another tutorial`), but an additional node update must be performed when the unknowns are updated by the Newton solver. This is achieved by performing a further node-update in the function `actions_before_↩ newton_convergence_check()`:

```
 /// 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()
  {
   Fluid_mesh_pt->node_update();
  }
```

The private member data contains the usual pointers to the `Problem's` two sub-meshes, the pointer to `Geom↩ Object` that represents the leaflet, and the total height of the channel.

```
private:

 /// Pointer to the fluid mesh
 RefineableAlgebraicChannelWithLeafletMesh<ELEMENT>* Fluid_mesh_pt;

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

 /// Pointer to the GeomObject that represents the wall
 GeomObject* Leaflet_pt;

 /// Total height of the domain
 double Htot;
};
```

1.7 The problem constructor

We start the problem construction by creating two timesteppers: A `BFD<2>` timestepper for the fluid and the fake timestepper `Steady<2>` for the (massless) solid. (`Recall` that timesteppers from the `Steady` family

return zero time-derivatives but keep track of the past histories. These are needed during the adaptive refinement of the fluid mesh: when assigning the history of the previous nodal positions for newly created fluid nodes, we must evaluate the position of the leaflet at previous timesteps; this is discussed in more detail in another tutorial.)

```
//=====start_of_constructor=======================================================
/// Constructor
//================================================================================
template <class ELEMENT>
FSIChannelWithLeafletProblem<ELEMENT>::FSIChannelWithLeafletProblem(
 const double& lleft,
 const double& lright,
 const double& hleaflet,
 const double& htot,
 const unsigned& nleft,
 const unsigned& nright,
 const unsigned& ny1,
 const unsigned&  ny2,
 const double& x_0) : Htot(htot)
{
 // Timesteppers:
 //--------------
 // Allocate the timestepper
 BDF<2>* fluid_time_stepper_pt=new BDF<2>;
 add_time_stepper_pt(fluid_time_stepper_pt);
 // Allocate the wall timestepper
 Steady<2>* wall_time_stepper_pt=new Steady<2>;
 add_time_stepper_pt(wall_time_stepper_pt);
```

Next we create the mesh of 1D Hermite beam elements that represents the leaflet, passing a pointer to an instance of the UndeformedLeaflet to its constructor.

```
 // Discretise leaflet
 //-------------------
 // Geometric object that represents the undeformed leaflet
 UndeformedLeaflet* undeformed_wall_pt=new UndeformedLeaflet(x_0);
 //Create the "wall" mesh with FSI Hermite beam elements
 unsigned n_wall_el=5;
 Wall_mesh_pt = new OneDLagrangianMesh<FSIHermiteBeamElement>
  (n_wall_el,hleaflet,undeformed_wall_pt,wall_time_stepper_pt);
```

The wall mesh defines the geometry of the deformed leaflet, therefore we create a GeomObject representation of that mesh, using the MeshAsGeomObject class, and pass it to the constructor of the fluid mesh:

```
 // Provide GeomObject representation of leaflet mesh and build fluid mesh
 //------------------------------------------------------------------------
 // Build a geometric object (one Lagrangian, two Eulerian coordinates)
 // from the wall mesh
 MeshAsGeomObject* wall_geom_object_pt=
  new MeshAsGeomObject(Wall_mesh_pt);
//Build the mesh
 Fluid_mesh_pt =new RefineableAlgebraicChannelWithLeafletMesh<ELEMENT>(
  wall_geom_object_pt,
  lleft, lright,
  hleaflet,
  htot,nleft,
  nright,ny1,ny2,
  fluid_time_stepper_pt);
```

We specify an error estimator for the fluid mesh and add both meshes to the Problem before combining them into a global mesh.

```
 // Set error estimator
 Z2ErrorEstimator* error_estimator_pt=new Z2ErrorEstimator;
 Fluid_mesh_pt->spatial_error_estimator_pt()=error_estimator_pt;
 // Build global mesh
 //------------------

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

Dirichlet conditions (prescribed velocity) are applied on virtually all boundaries of the fluid domain (prescribed inflow at the inlet; zero velocity on the rigid channel walls; fluid velocity prescribed by the wall motion on the leaflet), apart from the outflow (boundary 1; see the enumeration of the mesh boundaries in the ChannelWithLeafletMesh) where the axial velocity is unknown (and determined indirectly by the "axially-traction-free" condition) while the vertical velocity has to be set to zero to ensure parallel outflow.

```
 // Fluid boundary conditions
 //--------------------------
 //Pin the boundary nodes of the fluid mesh
 unsigned num_bound = Fluid_mesh_pt->nboundary();
 for(unsigned ibound=0;ibound<num_bound;ibound++)
  {
   unsigned num_nod= Fluid_mesh_pt->nboundary_node(ibound);
   for (unsigned inod=0;inod<num_nod;inod++)
    {
```

```
      Fluid_mesh_pt->boundary_node_pt(ibound,inod)->pin(1);

      // Do not pin the x velocity of the outflow
      if( ibound != 1)
      {
       Fluid_mesh_pt->boundary_node_pt(ibound,inod)->pin(0);
      }
    }
  }
 // end loop over boundaries
```

Next we assign the parabolic velocity profile at the inlet (recall that all values are initialised to zero so no further action is required on any of the other Dirichlet boundaries).

```
// Setup parabolic flow along boundary 3 (everything else that's
// pinned has homogenous boundary conditions so no action is required
// as that's the default assignment). Inflow profile is parabolic
// and this is interpolated correctly during mesh refinement so
// no re-assignment necessary after adaptation.
unsigned ibound=3;
unsigned num_nod= Fluid_mesh_pt->nboundary_node(ibound);
for (unsigned inod=0;inod<num_nod;inod++)
 {
  double ycoord = Fluid_mesh_pt->boundary_node_pt(ibound,inod)->x(1);
  double uy = 6.0*ycoord/htot*(1.0-ycoord/htot);
  Fluid_mesh_pt->boundary_node_pt(ibound,inod)->set_value(0,uy);
  Fluid_mesh_pt->boundary_node_pt(ibound,inod)->set_value(1,0.0);
 }// end of setup boundary condition
```

The leaflet is clamped at its lower end so we pin its x_1- and x_2-positions, and impose a vertical slope by setting $dx_1/ds = 0$ (where s is the local coordinate along the element; see the discussion of the boundary conditions for beam elements in another tutorial for details).

```
// Boundary conditions for wall mesh
//----------------------------------
// Set the boundary conditions: the lower end of the beam is fixed in space
unsigned b=0;
// 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);

// Infinite slope: Pin type 1 (slope) dof for displacement direction 0
wall_mesh_pt()->boundary_node_pt(b,0)->pin_position(1,0);
```

Next, we complete the build of the fluid elements by passing pointers to the relevant physical parameters to the elements and pinning any redundant pressure degrees of freedom; see another tutorial for details.

```
// Complete build of fluid elements
//---------------------------------
unsigned n_element = Fluid_mesh_pt->nelement();
// Loop over the elements to set up element-specific
// things that cannot be handled by constructor
for(unsigned e=0;e<n_element;e++)
 {
  // Upcast from GeneralisedElement to the present element
  ELEMENT* el_pt = dynamic_cast<ELEMENT*>(Fluid_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
// Pin redudant pressure dofs
RefineableNavierStokesEquations<2>::
 pin_redundant_nodal_pressures(Fluid_mesh_pt->element_pt());
```

When completing the build of the wall elements we use the function `enable_fluid_loading_on_both_sides()` to indicate that the leaflet is completely immersed in the fluid so that fluid tractions act on both of its faces. When setting up the fluid-structure interaction below, one of the two faces will have to be identified as the "front" (face 0) and the other one as the "back" (face 1). The function `normal_points_into_fluid()` allows us to indicate if the outer unit normal on the leaflet (as computed by `FSIHermiteBeamElement::get_normal(...)` points into the fluid, when viewed from the "front" face. Here it does not – see Further comments for more details on this slightly subtle point).

```
// Complete build of wall elements
//--------------------------------
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->h_pt() = &Global_Physical_Variables::H;

   // 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;
   // Leaflet is immersed and loaded by fluid on both sides
   elem_pt->enable_fluid_loading_on_both_sides();
   // The normal to the leaflet, as computed by the
   // FSIHermiteElements points away from the fluid rather than
   // into the fluid (as assumed by default) when viewed from
   // the "front" (face 0).
   elem_pt->set_normal_pointing_out_of_fluid();
  } // end of loop over elements
```

We can now set up the fluid structure interaction. The motion of the leaflet determines the fluid velocity of all nodes on boundaries 4 and 5 via the no-slip condition (see the enumeration of the mesh boundaries in the `ChannelWithLeafletMesh`). The fluid velocity at these nodes can be updated automatically whenever a fluid node is moved (by the fluid mesh's node-update function) by specifying an auxiliary node update function.

```
 // Setup FSI
 //----------

 // The velocity of the fluid nodes on the wall (fluid mesh boundary 4,5)
 // 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:
 for(unsigned ibound=4;ibound<6;ibound++ )
  {
   unsigned num_nod= Fluid_mesh_pt->nboundary_node(ibound);
   for (unsigned inod=0;inod<num_nod;inod++)
    {
     Fluid_mesh_pt->boundary_node_pt(ibound, inod)->
      set_auxiliary_node_update_fct_pt(
       FSI_functions::apply_no_slip_on_moving_wall);
    }
  }// aux node update fct has been set
```

Finally, we have to determine which fluid elements are adjacent to the two faces of the `FSIHermiteBeamElements` to allow them to compute the fluid traction they are exposed to. This is done separately for the "front" and "back" faces. The "front" of the leaflet (face 0) is assumed to coincide with the fluid mesh boundary 4; the "back" (face 1) is assumed to coincide with the fluid mesh boundary 5.

```
 // 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 4 and 5
 // of the 2D fluid mesh.
 // Front of leaflet: Set face=0 (which is also the default so this argument
 // could be omitted)
 unsigned face=0;
 FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
  (this,4,Fluid_mesh_pt,Wall_mesh_pt,face);

 // Back of leaflet: face 1, needs to be specified explicitly
 face=1;
 FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
  (this,5,Fluid_mesh_pt,Wall_mesh_pt,face);
```

Following the setup of the fluid-structure interaction we assign the equation numbers:

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

1.8 Actions after the mesh adaptation

Once the fluid mesh has been adapted we free all pressure degrees of freedom and then (re-)pin any redundant ones; see the discussion in `another tutorial` for details.

```
//==== start_of_actions_after_adapt=================================
/// Actions_after_adapt()
//==================================================================
template<class ELEMENT>
void FSIChannelWithLeafletProblem<ELEMENT>::actions_after_adapt()
{
 // Unpin all pressure dofs
 RefineableNavierStokesEquations<2>::
  unpin_all_pressure_dofs(Fluid_mesh_pt->element_pt());

 // Pin redundant pressure dofs
 RefineableNavierStokesEquations<2>::
  pin_redundant_nodal_pressures(Fluid_mesh_pt->element_pt());
```

Any newly created fluid nodes on the leaflet (i.e. on the fluid mesh boundaries 4 and 5) must be subjected to the automatic application of the no-slip condition. For simplicity we (re-)specify the auxiliary node update function pointer for all of the nodes on those boundaries.

```
 // (Re-)apply the no slip condition on the moving wall
 //----------------------------------------------------
 // The velocity of the fluid nodes on the wall (fluid mesh boundary 4,5)
 // 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:
 for(unsigned ibound=4;ibound<6;ibound++ )
  {
   unsigned num_nod= Fluid_mesh_pt->nboundary_node(ibound);
   for (unsigned inod=0;inod<num_nod;inod++)
    {
     Fluid_mesh_pt->boundary_node_pt(ibound, inod)->
      set_auxiliary_node_update_fct_pt(
       FSI_functions::apply_no_slip_on_moving_wall);
    }
  } // aux node update fct has been (re-)set
```

Finally, the adaptation of the fluid mesh may change the fluid elements that are adjacent to the wall elements so we re-generate the corresponding FSI lookup schemes.

```
 // Re-setup FSI
 //-------------

 // 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 4 and 5
 // of the 2D fluid mesh.
 // Front of leaflet: Set face=0 (which is also the default so this argument
 // could be omitted)
 unsigned face=0;
 FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
  (this,4,Fluid_mesh_pt,Wall_mesh_pt,face);

 // Back of leaflet: face 1, needs to be specified explicitly
 face=1;
 FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
  (this,5,Fluid_mesh_pt,Wall_mesh_pt,face);

} // end_of_actions_after_adapt
```

1.9 Post Processing

The function `doc_solution(...)` documents the results. We output the fluid and solid meshes and document selected additional quantities in the trace file.

```
//==start_of_doc_solution=============================================
/// Doc the solution
//====================================================================
template<class ELEMENT>
void FSIChannelWithLeafletProblem<ELEMENT>::doc_solution(DocInfo& doc_info,
                                                         ofstream& trace)
{
 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);
 Fluid_mesh_pt->output(some_file,npts);
 some_file.close();
 // Output wall solution
 sprintf(filename,"%s/wall_soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Wall_mesh_pt->output(some_file,npts);
 some_file.close();
 // Get node at tip of leaflet
 unsigned n_el_wall=Wall_mesh_pt->nelement();
 Node* tip_node_pt=Wall_mesh_pt->finite_element_pt(n_el_wall-1)->node_pt(1);
 // Get time:
 double time=time_pt()->time();
 // Write trace file
 trace << time << " "
       << Global_Physical_Variables::flux(time) << " "
       << tip_node_pt->x(0) << " "
```

```
       << tip_node_pt->x(1) << " "
       << tip_node_pt->dposition_dt(0) << " "
       << tip_node_pt->dposition_dt(1) << " "
       << doc_info.number() << " "
       << std::endl;
```

The remaining output is created to determine which elements are located next to the fluid mesh boundaries 4 and 5 (Yes, the documentation for the mesh illustrates the enumeration of the mesh boundaries but we generally prefer to be paranoid; see Further comments), and to establish the direction of the unit normal on the beam.

```
 // Help me figure out what the "front" and "back" faces of the leaflet are
 //-----------------------------------------------------------------------
 // Output fluid elements on fluid mesh boundary 4 (associated with
 // the "front")
 unsigned bound=4;
 sprintf(filename,"%s/fluid_boundary_elements_front_%i.dat",
         doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 unsigned nel= Fluid_mesh_pt->nboundary_element(bound);
 for (unsigned e=0;e<nel;e++)
  {
   dynamic_cast<ELEMENT*>(Fluid_mesh_pt->boundary_element_pt(bound,e))
    ->output(some_file,npts);
  }
 some_file.close();
 // Output fluid elements on fluid mesh boundary 5 (associated with
 // the "back")
 bound=5;
 sprintf(filename,"%s/fluid_boundary_elements_back_%i.dat",
         doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 nel= Fluid_mesh_pt->nboundary_element(bound);
 for (unsigned e=0;e<nel;e++)
  {
   dynamic_cast<ELEMENT*>(Fluid_mesh_pt->boundary_element_pt(bound,e))
    ->output(some_file,npts);
  }
 some_file.close();
 // Output normal vector on wall elements
 sprintf(filename,"%s/wall_normal_%i.dat",
         doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 nel=Wall_mesh_pt->nelement();
 Vector<double> s(1);
 Vector<double> x(2);
 Vector<double> xi(1);
 Vector<double> N(2);
 for (unsigned e=0;e<nel;e++)
  {
   // Get pointer to element
   FSIHermiteBeamElement* el_pt=
    dynamic_cast<FSIHermiteBeamElement*>(Wall_mesh_pt->element_pt(e));
   // Loop over plot points
   for (unsigned i=0;i<npts;i++)
    {
     s[0]=-1.0+2.0*double(i)/double(npts-1);
     // Get Eulerian position
     el_pt->interpolated_x(s,x);
     // Get unit normal
     el_pt->get_normal(s,N);
     // Get Lagrangian coordinate
     el_pt->interpolated_xi(s,xi);

     some_file << x[0] << " " << x[1] << " "
               << N[0] << " " << N[1] << " "
               << xi[0] << std::endl;
    }
  }
 some_file.close();
} // end_of_doc_solution
```

1.10 Further comments and exercises

1.10.1 Further comments

1. **How does one identify the "front" and "back" of an immersed beam structure?**

   When setting up the fluid-structure interaction in the The problem constructor we had to associate the fluid mesh boundaries 4 and 5 (the boundaries adjacent to the leaflet (see the enumeration of the mesh

boundaries in the `ChannelWithLeafletMesh`) with the "front" and "back" of the fully immersed beam structure. Furthermore, since the computation of the fluid traction acting on the leaflet depends on the direction of the normal, and the normal on the "front" of the leaflet points in the opposite direction to that on its "back" it is important to assess which normal is used in the automatic computation of the fluid traction. There are two ways of establishing this:

(a) **Quick and dirty:**
Set `FSIHermiteBeamElement::normal_points_into_fluid()` to `true` and perform a steady computation. If the leaflet is sucked towards the high-pressure region you got it wrong.

(b) **Do it properly:**
Use the output generated in `doc_solution(...)` to identify the elements adjacent to the fluid mesh boundaries 4 and 5 (i.e. the fluid elements next to the leaflet's "front" and "back" faces), respectively, and the normal vector returned by `FSIHermiteBeamElement::get_normal(...)` – this is the normal that is used in the computation of the fluid traction. The figure below shows a plot of these within an adaptively refined fluid mesh.

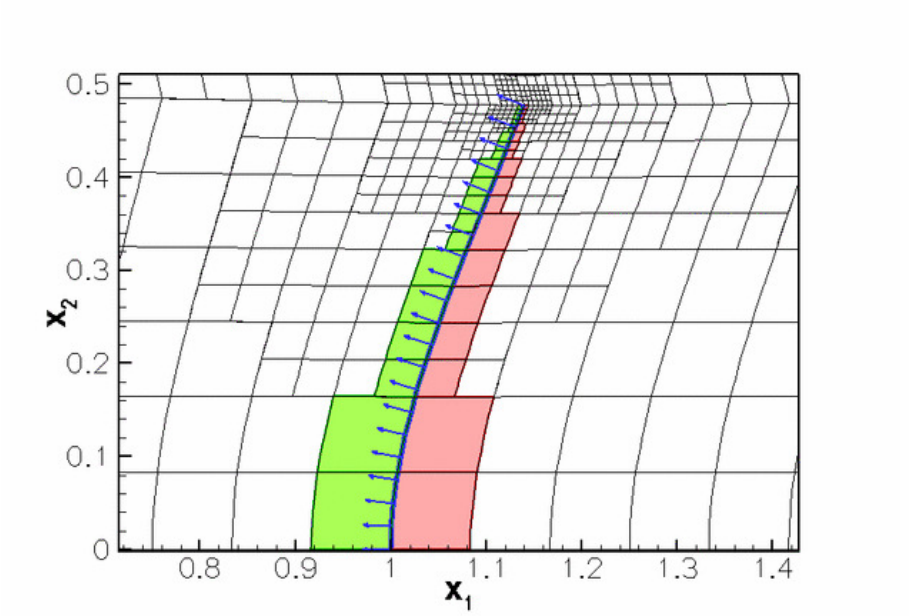

Figure 1.4 Elements near the `front' (red) and `back' (green) of the leaflet and the outer unit normal on the leaflet.

The red and green elements are the fluid elements adjacent to the fluid mesh boundaries 4 and 5 (i.e. the "front" and "back" of the leaflet), respectively. The normal vectors (shown in blue) point towards the upstream end of the channel so they point *away from* (rather than *into*) the fluid that is adjacent to the "front" (the red region). This is why we set `FSIHermiteBeamElement::normal_points_into_fluid()=false` (Admittedly, we used the former method to get this right and only documented the proper way to do it when writing this tutorial...)

2. **Use of faster solvers: `oomph-lib's` FSI preconditioner**

The problem presented in this tutorial was used as one of the test cases for `oomph-lib's` FSI preconditioner; see

Heil, M., Hazel, A.L. & Boyle, J. (2008): Solvers for large-displacement fluid-structure interaction problems: Segregated vs. monolithic approaches. Computational Mechanics.

The use of this preconditioner (which leads to much faster solution times) is described in another tutorial. However, the required source code is already part of the driver code and has simply been ignored in this tutorial. Feel free to experiment with the preconditioner by modifying the source code.

1.10.2 Exercises

1. Confirm that choosing the wrong value for the flag `FSIHermiteBeamElement::normal_points_into_fluid()` makes the leaflet move in the wrong direction as it perceives positive fluid tractions as negative ones and vice versa. Here is what you should get:

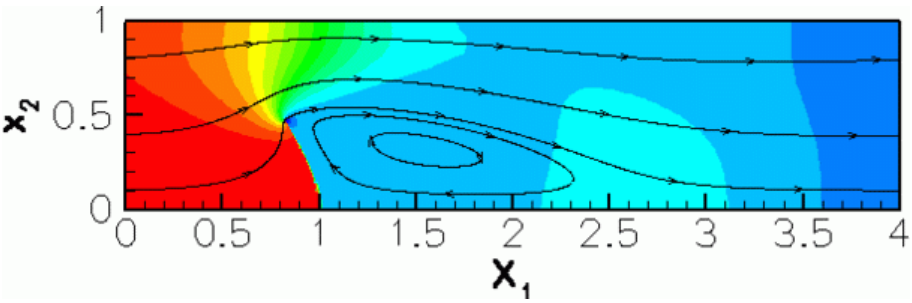

Figure 1.5 Deformation of the leaflet with the wrong choice of FSIHermiteBeamElement::normal_points_into_fluid(): The leaflet gets sucked towards the high-pressure region.

Clearly, the normal points in the wrong direction and the tractions have the wrong sign. This can be made "consistent", however, by also swapping the association between the mesh boundaries and the leaflet's faces. I.e. if we associate mesh boundary 5 with the "front" via

```
// Front of leaflet: face 0
face=0;
FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
 (this,5,Fluid_mesh_pt,Wall_mesh_pt,face);
```

and mesh boundary 4 with the "back" via

```
// Back of leaflet: face 1
face=1;
FSI_functions::setup_fluid_load_info_for_solid_elements<ELEMENT,2>
 (this,4,Fluid_mesh_pt,Wall_mesh_pt,face);
```

the code gives the right results again (make sure you change the code in the problem constructor and in the actions after adaptation!) – sometimes two wrongs do cancel each other out!

2. The animation of the flow field shows that the outflow boundary condition is applied "too close" to the leaflet: in the relatively short domain chosen here, the outflow is far from fully developed and as a result the imposition of a parallel flow leads to the development of a artificial outflow boundary layer.

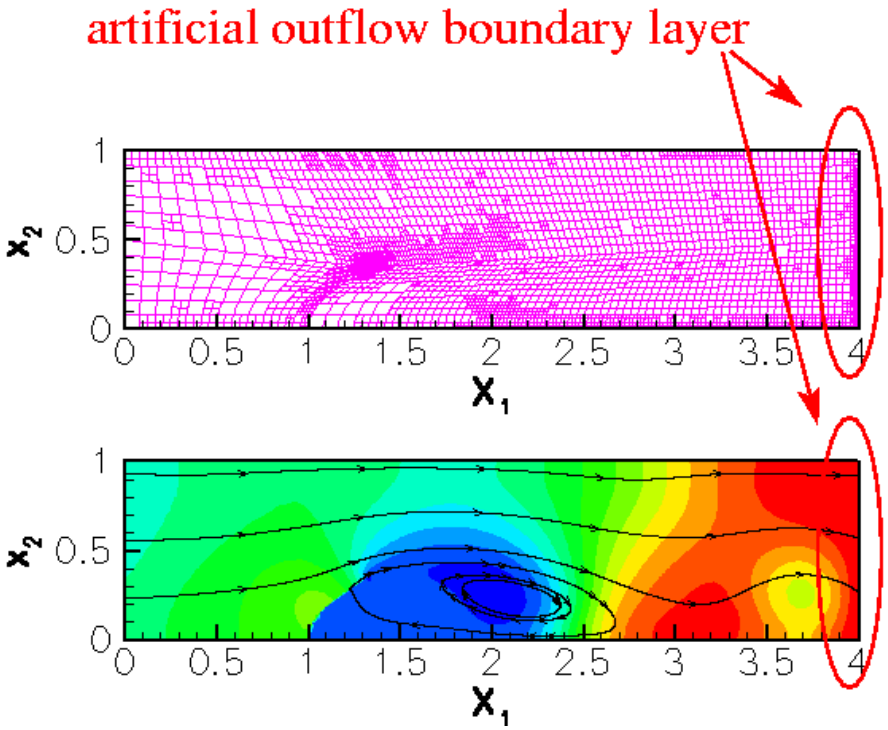

Figure 1.6 Artificial outflow boundary layer.

Confirm that an increase in the downstream length of the domain (or a reduction in the Reynolds number) improves the situation.

3. The algebraic node-update technique employed in the `ChannelWithLeafletMesh` forces *all* fluid nodes to move when the leaflet deforms. An increase in the downstream length of the fluid domain therefore leads to a significant increase in computational cost. Use the improved node-update technique suggested in the `corresponding Navier-Stokes example` (only move the nodes in the vicinity of the leaflet) to improve the efficiency of the code.

1.11 Acknowledgements

- This code was originally developed by Floraine Cordier.

1.12 Source files for this tutorial

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

`demo_drivers/interaction/fsi_channel_with_leaflet/`

- The driver code is:

demo_drivers/interaction/fsi_channel_with_leaflet/fsi_channel_with_↩
leaflet.cc

1.13 PDF file

A pdf version of this document is available.