Chapter 1

Time-harmonic acoustic fluid-structure interaction problems on unstructured meshes

In this document we discuss the solution of time-harmonic acoustic fluid-structure interaction problems on unstructured meshes.

The driver code is very similar to the one presented in another tutorial and we only discuss the changes necessary to deal with the generation of the unstructured mesh for the solid domain and the assignment of different material properties to different parts of the domain.

1.1 A test problem

The sketch below shows the problem setup: A 2D elastic annulus which is reinforced with two T-ribs is immersed in a compressible fluid and subjected to a time-periodic pressure load of magnitude

$$\mathbf{t} = P(\exp(\alpha(\varphi - \pi/4)^2) + \exp(\alpha(\varphi - 3\pi/4)^2))$$

(where φ is the polar angle) along its inner surface. The parameter α controls the "sharpness" of the pressure load. For $\alpha = 0$ we obtain a uniform, axisymmetric load; the sketch below shows the pressure distribution (red vectors indicating the traction) for $\alpha = 200$.

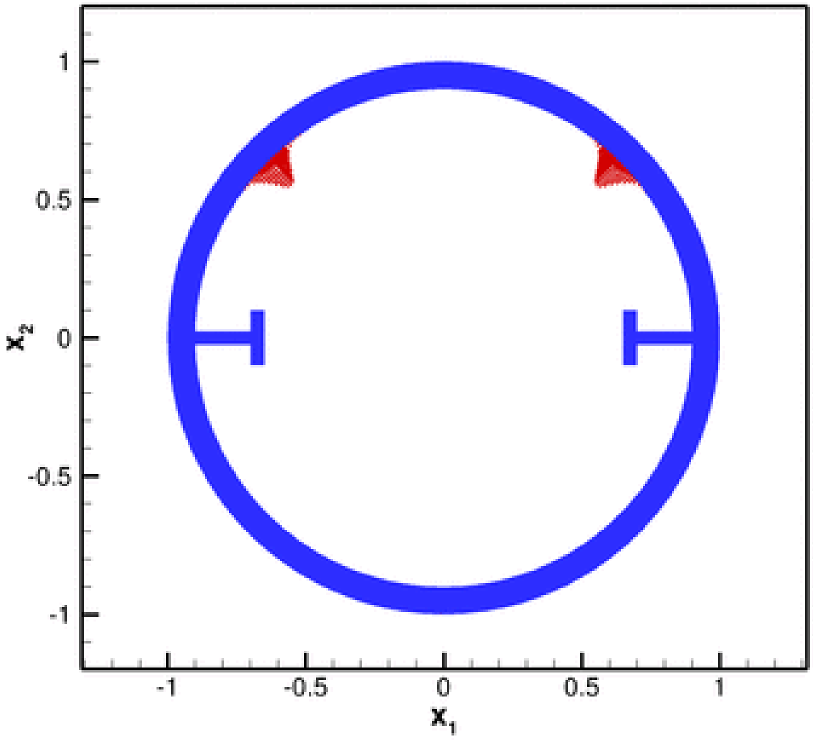

Figure 1.1 Sketch of the problem setup.

The structure is symmetric and we only discretise the right half ($x_1 > 0$) of the domain and apply symmetry conditions (zero horizontal displacement) on the x_2- axis.

1.2 Results

The figure below shows an animation of the structure's time-harmonic oscillation. The blue shaded region shows the shape of the oscillating structure while the pink region shows its initial configuration. The left half of the plot is used to show the (mirror image of the) adaptive unstructured mesh on which the solution was computed:

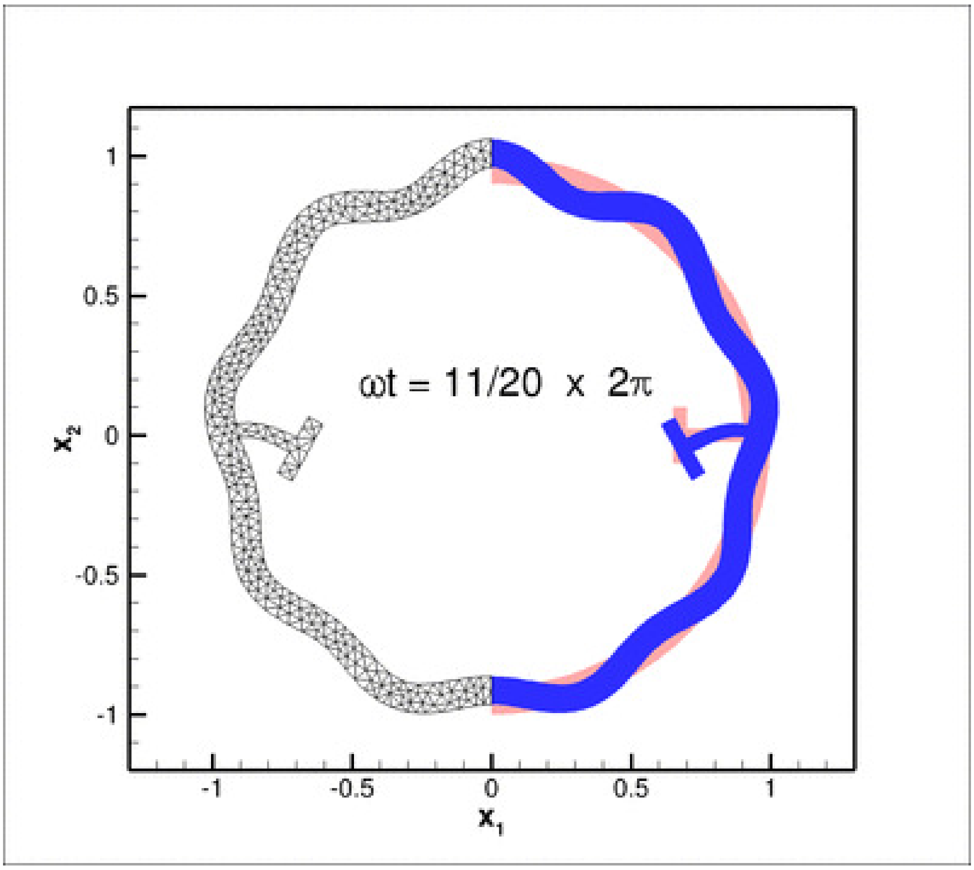

Figure 1.2 Animation of the time-harmonic deformation.

Here is a plot of the corresponding fluid displacement potential, a measure of the fluid pressure:

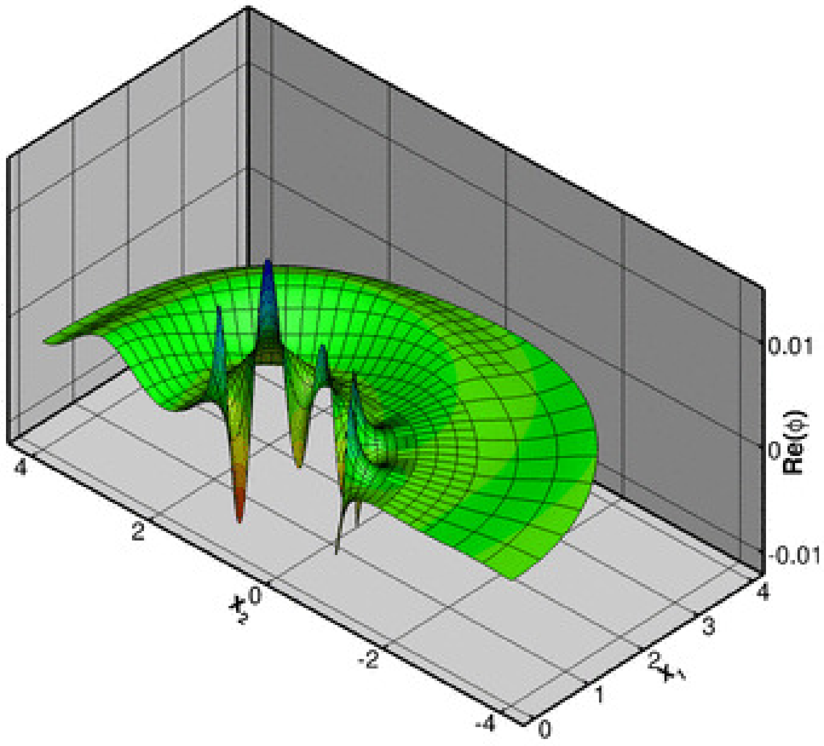

Figure 1.3 The fluid displacement potential, a measure of the fluid pressure. Elevation: real part; contours: imaginary part.

This looks very pretty and shows that we can solve acoustic FSI problems in non-trivial geometries but should you believe the results? Here's an attempt to convince you: If we make the rib much softer than the annulus and set its inertia to zero the rib will not offer much structural resistance and the annular region will deform as if the rib was not present. If we then set $\alpha = 0$ we apply an axisymmetric forcing onto the structure and would expect the resulting displacement field (at least in the annular region) to be axisymmetric.

The animation of the displacement field for this case, shown below, shows that this is indeed the case:

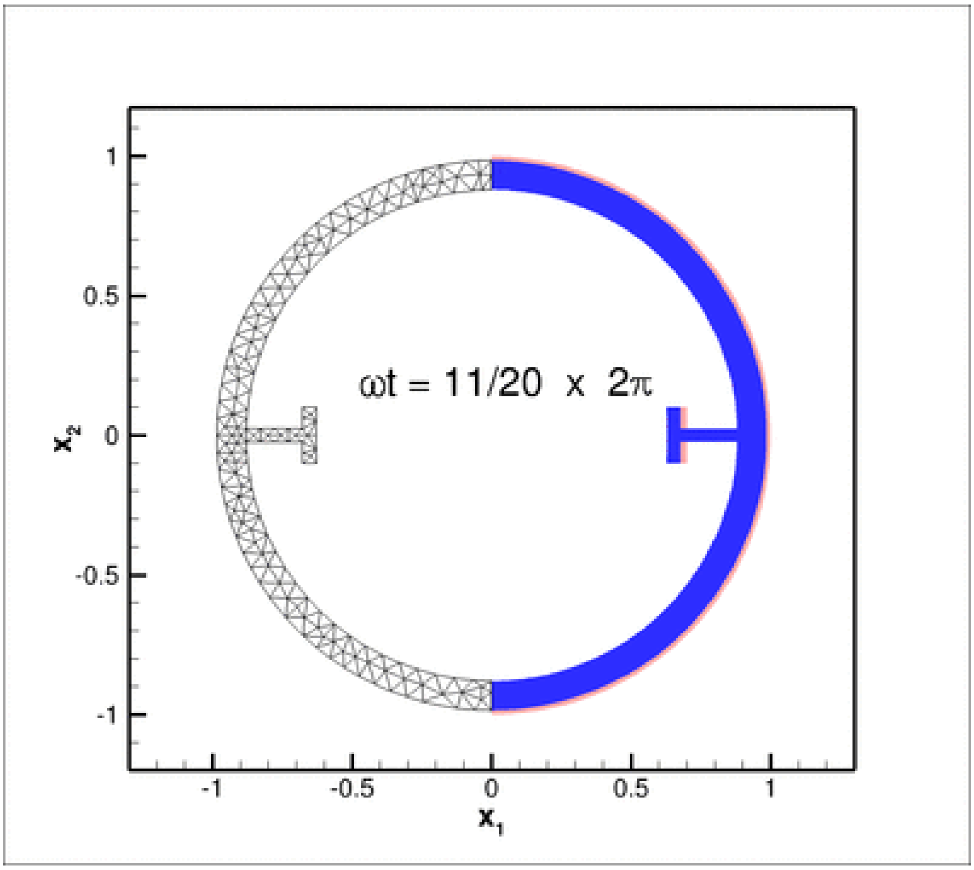

Figure 1.4 Animation of the time-harmonic deformation for uniform pressure load and a very soft and inertia-less rib.

Here is a plot of the corresponding fluid displacement potential, a measure of the fluid pressure:

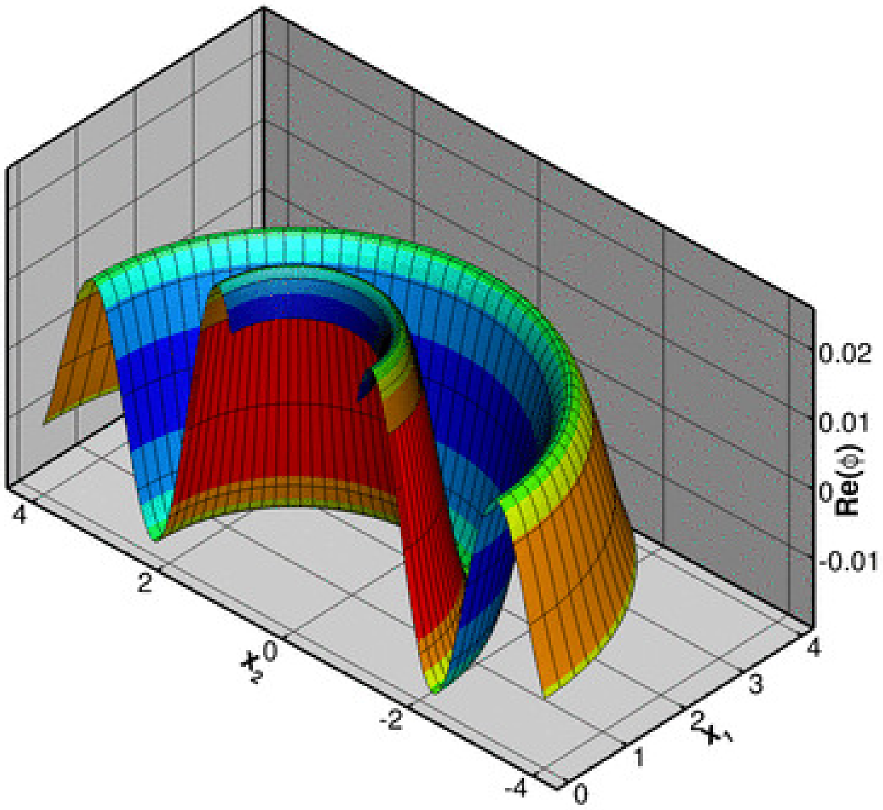

Figure 1.5 The fluid displacement potential, a measure of the fluid pressure for a uniform pressure load and very soft and inertia-less rib. Elevation: real part; contours: imaginary part.

1.3 The numerical solution

The driver code for this problem is very similar to the one discussed in `another tutorial`. Running `sdiff` on the two driver codes

`demo_drivers/interaction/acoustic_fsi/acoustic_fsi.cc`

and

`demo_drivers/interaction/acoustic_fsi/unstructured_acoustic_fsi.cc`

shows you the differences, the most important of which are:

- The use of `approximate/absorbing boundary conditions (ABCs)` rather than a `Dirichlet-to-Neumann mapping` for the Helmholtz equation, because the boundary along which the Sommerfeld radiation condition is applied is not a full circle.

- The provision of multiple elasticity tensors and frequency parameters for the two different regions (the rib and the annulus).

- The change of forcing from a prescribed time-harmonic displacement to a pressure load on the inside boundary – this requires yet another mesh of `FaceElements`.

- The provision of a helper function `complete_problem_setup()` which rebuilds the elements (by passing the problem parameters to the elements) following the unstructured mesh adaptation. (The need/rationale for such a function is discussed in another tutorial.)

- The mesh generation – the specification of the curvilinear boundaries and the geometry of the rib is somewhat tedious. We refer to another tutorial for a discussion of how to define the internal mesh boundary that separates the two regions (the rib and the annular region) so that we can assign different material properties to them.

All of this is reasonably straightforward and provides a powerful code that automatically adapts both meshes while respecting the curvilinear boundaries of the domain. Have a look through the driver code and play with it.

1.4 Code listing

Here's a listing of the complete driver code:

```
//LIC// ====================================================================
//LIC// This file forms part of oomph-lib, the object-oriented,
//LIC// multi-physics finite-element library, available
//LIC// at http://www.oomph-lib.org.
//LIC//
//LIC//    Version 1.0; svn revision $LastChangedRevision$
//LIC//
//LIC// $LastChangedDate$
//LIC//
//LIC// Copyright (C) 2006-2016 Matthias Heil and Andrew Hazel
//LIC//
//LIC// This library is free software; you can redistribute it and/or
//LIC// modify it under the terms of the GNU Lesser General Public
//LIC// License as published by the Free Software Foundation; either
//LIC// version 2.1 of the License, or (at your option) any later version.
//LIC//
//LIC// This library is distributed in the hope that it will be useful,
//LIC// but WITHOUT ANY WARRANTY; without even the implied warranty of
//LIC// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
//LIC// Lesser General Public License for more details.
//LIC//
//LIC// You should have received a copy of the GNU Lesser General Public
//LIC// License along with this library; if not, write to the Free Software
//LIC// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
//LIC// 02110-1301  USA.
//LIC//
//LIC// The authors may be contacted at oomph-lib@maths.man.ac.uk.
//LIC//
//LIC//====================================================================
// Driver for Helmholtz/TimeHarmonicTimeHarmonicLinElast coupling
//Oomph-lib includes
#include "generic.h"
#include "helmholtz.h"
#include "time_harmonic_linear_elasticity.h"
#include "multi_physics.h"
//The meshes
#include "meshes/annular_mesh.h"
#include "meshes/triangle_mesh.h"
using namespace std;
using namespace oomph;

///////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////
// Straight line as geometric object
///////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////

//=====================================================================
/// Straight 1D line in 2D space
//=====================================================================
class MyStraightLine : public GeomObject
{
public:

 /// Constructor:  Pass start and end points
 MyStraightLine(const Vector<double>& r_start, const Vector<double>& r_end)
  :  GeomObject(1,2), R_start(r_start), R_end(r_end)
  { }
```

```
 /// Broken copy constructor
 MyStraightLine(const MyStraightLine& dummy)
  {
   BrokenCopy::broken_copy("MyStraightLine");
  }

 /// Broken assignment operator
 void operator=(const MyStraightLine&)
  {
   BrokenCopy::broken_assign("MyStraightLine");
  }

 /// Destructor:  Empty
 ~MyStraightLine(){}

 /// \short Position Vector at Lagrangian coordinate zeta
 void position(const Vector<double>& zeta, Vector<double>& r) const
  {
   // Position Vector
   r[0] = R_start[0]+(R_end[0]-R_start[0])*zeta[0];
   r[1] = R_start[1]+(R_end[1]-R_start[1])*zeta[0];
  }

private:

 /// Start point of line
 Vector<double> R_start;

 /// End point of line
 Vector<double> R_end;
};

//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

//=======start_namespace==========================================
/// Global variables
//================================================================
namespace Global_Parameters
{

 /// \short Square of wavenumber for the Helmholtz equation
 double K_squared=10.0;

 /// \short Radius of outer boundary of Helmholtz domain
 double Outer_radius=4.0;

 /// Order of absorbing/appproximate boundary condition
 unsigned ABC_order=3;

 /// FSI parameter
 double Q=0.0;

 /// Non-dim thickness of elastic coating
 double H_coating=0.1;

 /// Poisson's ratio
 double Nu = 0.3;

 /// Square of non-dim frequency for the two regions -- dependent variable!
 Vector<double> Omega_sq(2,0.0);

 /// Density ratio for the two regions: solid to fluid
 Vector<double> Density_ratio(2,0.1);

 /// The elasticity tensors for the two regions
 Vector<TimeHarmonicIsotropicElasticityTensor*> E_pt;

 /// Function to update dependent parameter values
 void update_parameter_values()
 {
  Omega_sq[0]=Density_ratio[0]*Q;
  Omega_sq[1]=Density_ratio[1]*Q;
 }

 /// Uniform pressure
 double P = 0.1;

 /// Peakiness parameter for pressure load
 double Alpha=200.0;

 /// Pressure load (real and imag part)
 void pressure_load(const Vector<double> &x,
                    const Vector<double> &n,
                    Vector<std::complex<double> >&traction)
 {
```

```
  double phi=atan2(x[1],x[0]);
  double magnitude=exp(-Alpha*pow(phi-0.25*MathematicalConstants::Pi,2));
  unsigned dim = traction.size();
  for(unsigned i=0;i<dim;i++)
   {
    traction[i] = complex<double>(-magnitude*P*n[i],magnitude*P*n[i]);
   }
 } // end_of_pressure_load
 // Output directory
 string Directory="RESLT";

 // Multiplier for number of elements
 unsigned El_multiplier=1;
} //end namespace
//=============begin_problem============================================
/// Coated disk FSI
//======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
class CoatedDiskProblem : public Problem
{
public:

 /// Constructor:
 CoatedDiskProblem();

 /// Update function (empty)
 void actions_before_newton_solve() {}

 /// Update function (empty)
 void actions_after_newton_solve() {}

 /// Actions before adapt: Wipe the face meshes
 void actions_before_adapt();

 /// Actions after adapt: Rebuild the face meshes
 void actions_after_adapt();

 /// Doc the solution
 void doc_solution();
private:
 // Complete problem setup
 void complete_problem_setup();

 /// \short Create solid traction elements
 void create_solid_traction_elements();

 /// \short Create FSI traction elements
 void create_fsi_traction_elements();

 /// \short Create Helmholtz FSI flux elements
 void create_helmholtz_fsi_flux_elements();

 /// Delete (face) elements in specified mesh
 void delete_face_elements(Mesh* const & boundary_mesh_pt);

 /// \short Create ABC face elements
 void create_helmholtz_ABC_elements();

 /// Setup interaction
 void setup_interaction();

 /// Pointer to refineable solid mesh
 RefineableTriangleMesh<ELASTICITY_ELEMENT>* Solid_mesh_pt;

 /// Pointer to mesh of solid traction elements
 Mesh* Solid_traction_mesh_pt;

 /// Pointer to mesh of FSI traction elements
 Mesh* FSI_traction_mesh_pt;

 /// Pointer to Helmholtz mesh
 TreeBasedRefineableMeshBase* Helmholtz_mesh_pt;

 /// Pointer to mesh of Helmholtz FSI flux elements
 Mesh* Helmholtz_fsi_flux_mesh_pt;

 /// \short Pointer to mesh containing the ABC elements
 Mesh* Helmholtz_outer_boundary_mesh_pt;

 /// Boundary ID of upper symmetry boundary
 unsigned Upper_symmetry_boundary_id;

 /// Boundary ID of lower symmetry boundary
 unsigned Lower_symmetry_boundary_id;

 /// Boundary ID of upper inner boundary
 unsigned Upper_inner_boundary_id;
```

```
 /// Boundary ID of lower inner boundary
 unsigned Lower_inner_boundary_id;

 /// Boundary ID of outer boundary
 unsigned Outer_boundary_id;
 // Boundary ID of rib divider
 unsigned Rib_divider_boundary_id;

 /// DocInfo object for output
 DocInfo Doc_info;

 /// Trace file
 ofstream Trace_file;
};
//===========start_of_constructor=======================================
/// Constructor
//======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
CoatedDiskProblem<ELASTICITY_ELEMENT, HELMHOLTZ_ELEMENT>::CoatedDiskProblem()
{
 // To suppress boundary warnings (to avoid a lot of warnings) uncomment
 // the following code:
 //UnstructuredTwoDMeshGeometryBase::
 // Suppress_warning_about_regions_and_boundaries=true;
 // Solid mesh
 //-----------
 // Start and end coordinates
 Vector<double> r_start(2);
 Vector<double> r_end(2);

 // Outer radius of hull
 double r_outer = 1.0;
 // Inner radius of hull
 double r_inner = r_outer-Global_Parameters::H_coating;
 // Thickness of rib
 double rib_thick=0.05;

 // Depth of rib
 double rib_depth=0.2;
 // Total width of T
 double t_width=0.2;
 // Thickness of T
 double t_thick=0.05;
 // Half-opening angle of rib
 double half_phi_rib=asin(0.5*rib_thick/r_inner);
 // Pointer to the closed curve that defines the outer boundary
 TriangleMeshClosedCurve* closed_curve_pt=0;

 // Provide storage for pointers to the parts of the curvilinear boundary
 Vector<TriangleMeshCurveSection*> curvilinear_boundary_pt;
 // Outer boundary
 //---------------
 Ellipse* outer_boundary_circle_pt = new Ellipse(r_outer,r_outer);
 double zeta_start=-0.5*MathematicalConstants::Pi;
 double zeta_end=0.5*MathematicalConstants::Pi;
 unsigned nsegment=50;
 unsigned boundary_id=curvilinear_boundary_pt.size();
 curvilinear_boundary_pt.push_back(
  new TriangleMeshCurviLine(
   outer_boundary_circle_pt,zeta_start,zeta_end,nsegment,boundary_id));
 // Remember it
 Outer_boundary_id=boundary_id;

 // Upper straight line segment on symmetry axis
 //---------------------------------------------
 r_start[0]=0.0;
 r_start[1]=r_outer;
 r_end[0]=0.0;
 r_end[1]=r_inner;
 MyStraightLine* upper_sym_pt = new MyStraightLine(r_start,r_end);
 zeta_start=0.0;
 zeta_end=1.0;
 nsegment=1;
 boundary_id=curvilinear_boundary_pt.size();
 curvilinear_boundary_pt.push_back(
  new TriangleMeshCurviLine(
   upper_sym_pt,zeta_start,zeta_end,nsegment,boundary_id));

 // Remember it
 Upper_symmetry_boundary_id=boundary_id;

 // Upper part of inner boundary
 //-----------------------------
 Ellipse* upper_inner_boundary_pt =
  new Ellipse(r_inner,r_inner);
 zeta_start=0.5*MathematicalConstants::Pi;
```

```
zeta_end=half_phi_rib;
nsegment=20;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  upper_inner_boundary_pt,
  zeta_start,zeta_end,nsegment,boundary_id));
// Remember it
Upper_inner_boundary_id=boundary_id;
// Data associated with rib
MyStraightLine* upper_inward_rib_pt=0;
MyStraightLine* lower_inward_rib_pt=0;
TriangleMeshCurviLine* upper_inward_rib_curviline_pt=0;
Vector<TriangleMeshOpenCurve*> inner_boundary_pt;
TriangleMeshCurviLine* lower_inward_rib_curviline_pt=0;
Vector<double> rib_center(2);
// Upper half of inward rib
//-------------------------
r_start[0]=r_inner*cos(half_phi_rib);
r_start[1]=r_inner*sin(half_phi_rib);
r_end[0]=r_start[0]-rib_depth;
r_end[1]=r_start[1];
upper_inward_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
upper_inward_rib_curviline_pt=
 new TriangleMeshCurviLine(
  upper_inward_rib_pt,zeta_start,zeta_end,nsegment,boundary_id);
curvilinear_boundary_pt.push_back(upper_inward_rib_curviline_pt);
// Vertical upper bit of T
//------------------------
r_start[0]=r_end[0];
r_start[1]=r_end[1];
r_end[0]=r_start[0];
r_end[1]=r_start[1]+0.5*(t_width-rib_thick);
MyStraightLine* vertical_upper_t_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  vertical_upper_t_rib_pt,zeta_start,zeta_end,nsegment,boundary_id));
// Horizontal upper bit of T
//-----------==-------------
r_start[0]=r_end[0];
r_start[1]=r_end[1];
r_end[0]=r_start[0]-t_thick;
r_end[1]=r_start[1];
MyStraightLine* horizontal_upper_t_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  horizontal_upper_t_rib_pt,zeta_start,zeta_end,nsegment,boundary_id));
// Vertical end of rib end
//------------------------
r_start[0]=r_end[0];
r_start[1]=r_end[1];
r_end[0]=r_start[0];
r_end[1]=-r_start[1];
MyStraightLine* inner_vertical_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  inner_vertical_rib_pt,zeta_start,zeta_end,nsegment,boundary_id));

// Horizontal lower bit of T
//-----------==-------------
r_start[0]=r_end[0];
r_start[1]=r_end[1];
r_end[0]=r_start[0]+t_thick;
r_end[1]=r_start[1];
MyStraightLine* horizontal_lower_t_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
```

```
  horizontal_lower_t_rib_pt,zeta_start,zeta_end,nsegment,boundary_id));
// Vertical lower bit of T
//-------------------------
r_start[0]=r_end[0];
r_start[1]=r_end[1];
r_end[0]=r_start[0];
r_end[1]=r_start[1]+0.5*(t_width-rib_thick);
MyStraightLine* vertical_lower_t_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  vertical_lower_t_rib_pt,zeta_start,zeta_end,nsegment,boundary_id));
// Lower half of inward rib
//--------------------------
r_end[0]=r_inner*cos(half_phi_rib);
r_end[1]=-r_inner*sin(half_phi_rib);
r_start[0]=r_end[0]-rib_depth;
r_start[1]=r_end[1];
lower_inward_rib_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
lower_inward_rib_curviline_pt=
 new TriangleMeshCurviLine(
  lower_inward_rib_pt,zeta_start,zeta_end,nsegment,boundary_id);
curvilinear_boundary_pt.push_back(lower_inward_rib_curviline_pt);
// Lower part of inner boundary
//------------------------------
Ellipse* lower_inner_boundary_circle_pt = new Ellipse(r_inner,r_inner);
zeta_start=-half_phi_rib;
zeta_end=-0.5*MathematicalConstants::Pi;
nsegment=20;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  lower_inner_boundary_circle_pt,zeta_start,zeta_end,nsegment,boundary_id));

// Remember it
Lower_inner_boundary_id=boundary_id;
// Lower straight line segment on symmetry axis
//----------------------------------------------
r_start[0]=0.0;
r_start[1]=-r_inner;
r_end[0]=0.0;
r_end[1]=-r_outer;
MyStraightLine* lower_sym_pt = new MyStraightLine(r_start,r_end);
zeta_start=0.0;
zeta_end=1.0;
nsegment=1;
boundary_id=curvilinear_boundary_pt.size();
curvilinear_boundary_pt.push_back(
 new TriangleMeshCurviLine(
  lower_sym_pt,zeta_start,zeta_end,nsegment,boundary_id));

// Remember it
Lower_symmetry_boundary_id=boundary_id;
// Combine to curvilinear boundary
//---------------------------------
closed_curve_pt=
 new TriangleMeshClosedCurve(curvilinear_boundary_pt);

// Vertical dividing line across base of T-rib
//---------------------------------------------
Vector<TriangleMeshCurveSection*> internal_polyline_pt(1);
r_start[0]=r_inner*cos(half_phi_rib);
r_start[1]=r_inner*sin(half_phi_rib);
r_end[0]=r_inner*cos(half_phi_rib);
r_end[1]=-r_inner*sin(half_phi_rib);
Vector<Vector<double> > boundary_vertices(2);
boundary_vertices[0]=r_start;
boundary_vertices[1]=r_end;
boundary_id=100;
TriangleMeshPolyLine* rib_divider_pt=
 new TriangleMeshPolyLine(boundary_vertices,boundary_id);
internal_polyline_pt[0]=rib_divider_pt;
// Remember it
Rib_divider_boundary_id=boundary_id;
// Make connection
double s_connect=0.0;
internal_polyline_pt[0]->connect_initial_vertex_to_curviline(
 upper_inward_rib_curviline_pt,s_connect);
// Make connection
s_connect=1.0;
```

```
internal_polyline_pt[0]->connect_final_vertex_to_curviline(
 lower_inward_rib_curviline_pt,s_connect);
// Create open curve that defines internal bondary
inner_boundary_pt.push_back(new TriangleMeshOpenCurve(internal_polyline_pt));

// Define coordinates of a point inside the rib
rib_center[0]=r_inner-rib_depth;
rib_center[1]=0.0;
// Now build the mesh
//===================
// Use the TriangleMeshParameters object for helping on the manage of the
// TriangleMesh parameters. The only parameter that needs to take is the
// outer boundary.
TriangleMeshParameters triangle_mesh_parameters(closed_curve_pt);
// Target area
triangle_mesh_parameters.element_area()=0.2;
// Specify the internal open boundary
triangle_mesh_parameters.internal_open_curves_pt()=inner_boundary_pt;
// Define the region
triangle_mesh_parameters.add_region_coordinates(1,rib_center);

// Build the mesh
Solid_mesh_pt=new
 RefineableTriangleMesh<ELASTICITY_ELEMENT>(triangle_mesh_parameters);

// Helmholtz mesh
//---------------
// Number of elements in azimuthal direction in Helmholtz mesh
unsigned ntheta_helmholtz=11*Global_Parameters::El_multiplier;
// Number of elements in radial direction in Helmholtz mesh
unsigned nr_helmholtz=3*Global_Parameters::El_multiplier;
// Innermost radius of Helmholtz mesh
double a=1.0;

// Thickness of Helmholtz mesh
double h_thick_helmholtz=Global_Parameters::Outer_radius-a;
// Build mesh
bool periodic=false;
double azimuthal_fraction=0.5;
double phi=MathematicalConstants::Pi/2.0;
Helmholtz_mesh_pt = new
 RefineableTwoDAnnularMesh<HELMHOLTZ_ELEMENT>
 (periodic,azimuthal_fraction,
  ntheta_helmholtz,nr_helmholtz,a,h_thick_helmholtz,phi);
// Set error estimators
Solid_mesh_pt->spatial_error_estimator_pt()=new Z2ErrorEstimator;
Helmholtz_mesh_pt->spatial_error_estimator_pt()=new Z2ErrorEstimator;
// Mesh containing the Helmholtz ABC
// elements.
Helmholtz_outer_boundary_mesh_pt = new Mesh;
// Create elasticity tensors
Global_Parameters::E_pt.resize(2);
Global_Parameters::E_pt[0]=new TimeHarmonicIsotropicElasticityTensor(
   Global_Parameters::Nu);
Global_Parameters::E_pt[1]=new TimeHarmonicIsotropicElasticityTensor(
 Global_Parameters::Nu);
// Complete build of solid elements
complete_problem_setup();
// Same for Helmholtz mesh
unsigned n_element =Helmholtz_mesh_pt->nelement();
for(unsigned i=0;i<n_element;i++)
 {
  //Cast to a solid element
  HELMHOLTZ_ELEMENT *el_pt =
   dynamic_cast<HELMHOLTZ_ELEMENT*>(Helmholtz_mesh_pt->element_pt(i));
  //Set the pointer to square of Helmholtz wavenumber
  el_pt->k_squared_pt() = &Global_Parameters::K_squared;
 }
// Output meshes and their boundaries so far so we can double
// check the boundary enumeration
Solid_mesh_pt->output("solid_mesh.dat");
Helmholtz_mesh_pt->output("helmholtz_mesh.dat");
Solid_mesh_pt->output_boundaries("solid_mesh_boundary.dat");
Helmholtz_mesh_pt->output_boundaries("helmholtz_mesh_boundary.dat");
// Create FaceElement meshes for boundary conditions
//---------------------------------------------------
// Construct the solid traction element mesh
Solid_traction_mesh_pt=new Mesh;
create_solid_traction_elements();
// Construct the fsi traction element mesh
FSI_traction_mesh_pt=new Mesh;
create_fsi_traction_elements();
// Construct the Helmholtz fsi flux element mesh
Helmholtz_fsi_flux_mesh_pt=new Mesh;
create_helmholtz_fsi_flux_elements();
// Create ABC elements on outer boundary of Helmholtz mesh
create_helmholtz_ABC_elements();
```

```
 // Combine sub meshes
 //-------------------
 // Solid mesh is first sub-mesh
 add_sub_mesh(Solid_mesh_pt);
 // Add solid traction sub-mesh
 add_sub_mesh(Solid_traction_mesh_pt);
 // Add FSI traction sub-mesh
 add_sub_mesh(FSI_traction_mesh_pt);
 // Add Helmholtz mesh
 add_sub_mesh(Helmholtz_mesh_pt);
 // Add Helmholtz FSI flux mesh
 add_sub_mesh(Helmholtz_fsi_flux_mesh_pt);
 // Add Helmholtz ABC mesh
 add_sub_mesh(Helmholtz_outer_boundary_mesh_pt);
 // Build combined "global" mesh
 build_global_mesh();

 // Setup fluid-structure interaction
 //----------------------------------
 setup_interaction();
 // Assign equation  numbers
 oomph_info « "Number of unknowns: " « assign_eqn_numbers() « std::endl;
 // Set output directory
 Doc_info.set_directory(Global_Parameters::Directory);
 // Open trace file
 char filename[100];
 sprintf(filename,"%s/trace.dat",Doc_info.directory().c_str());
 Trace_file.open(filename);

} //end of constructor
//=====================start_of_actions_before_adapt======================
/// Actions before adapt: Wipe the meshes face elements
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
actions_before_adapt()
{
 // Kill the solid traction elements and wipe surface mesh
 delete_face_elements(Solid_traction_mesh_pt);
 // Kill the fsi traction elements and wipe surface mesh
 delete_face_elements(FSI_traction_mesh_pt);

 // Kill Helmholtz FSI flux elements
 delete_face_elements(Helmholtz_fsi_flux_mesh_pt);

 // Kill Helmholtz BC elements
 delete_face_elements(Helmholtz_outer_boundary_mesh_pt);

 // Rebuild the Problem's global mesh from its various sub-meshes
 rebuild_global_mesh();
}// end of actions_before_adapt
//=====================start_of_actions_after_adapt=======================
///  Actions after adapt: Rebuild the meshes of face elements
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
actions_after_adapt()
{
 // Complete problem setup
 complete_problem_setup();
 // Construct the solid traction elements
 create_solid_traction_elements();
 // Create fsi traction elements from all elements that are
 // adjacent to FSI boundaries and add them to surface meshes
 create_fsi_traction_elements();

 // Create Helmholtz fsi flux elements
 create_helmholtz_fsi_flux_elements();

 // Create ABC elements from all elements that are
 // adjacent to the outer boundary of Helmholtz mesh
 create_helmholtz_ABC_elements();

 // Setup interaction
 setup_interaction();

 // Rebuild the Problem's global mesh from its various sub-meshes
 rebuild_global_mesh();

}// end of actions_after_adapt
//=====================start_of_complete_problem_setup=======================
/// Complete problem setup: Apply boundary conditions and set
/// physical properties
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
complete_problem_setup()
```

```
{
 // Solid boundary conditions:
 //---------------------------
 // Pin real and imag part of horizontal displacement components
 //-------------------------------------------------------------
 // on vertical boundaries
 //-----------------------
 {
  //Loop over the nodes to pin and assign boundary displacements on
  //solid boundary
  unsigned n_node = Solid_mesh_pt->nboundary_node(Upper_symmetry_boundary_id);
  for(unsigned i=0;i<n_node;i++)
   {
    Node* nod_pt=Solid_mesh_pt->boundary_node_pt(Upper_symmetry_boundary_id,i);

    // Real part of x-displacement
    nod_pt->pin(0);
    nod_pt->set_value(0,0.0);

    // Imag part of x-displacement
    nod_pt->pin(2);
    nod_pt->set_value(2,0.0);
   }
 }
 {
  //Loop over the nodes to pin and assign boundary displacements on
  //solid boundary
  unsigned n_node = Solid_mesh_pt->nboundary_node(Lower_symmetry_boundary_id);
  for(unsigned i=0;i<n_node;i++)
   {
    Node* nod_pt=Solid_mesh_pt->boundary_node_pt(Lower_symmetry_boundary_id,i);
    // Real part of x-displacement
    nod_pt->pin(0);
    nod_pt->set_value(0,0.0);

    // Imag part of x-displacement
    nod_pt->pin(2);
    nod_pt->set_value(2,0.0);
   }
 }
 //Assign the physical properties to the elements
 //-----------------------------------------------
 unsigned nreg=Solid_mesh_pt->nregion();
 for (unsigned r=0;r<nreg;r++)
  {
   unsigned nel=Solid_mesh_pt->nregion_element(r);
   for (unsigned e=0;e<nel;e++)
    {
     //Cast to a solid element
     ELASTICITY_ELEMENT *el_pt =
      dynamic_cast<ELASTICITY_ELEMENT*>(Solid_mesh_pt->
                                         region_element_pt(r,e));

     // Set the constitutive law
     el_pt->elasticity_tensor_pt() = Global_Parameters::E_pt[r];

     // Square of non-dim frequency
     el_pt->omega_sq_pt()= &Global_Parameters::Omega_sq[r];
    }
  }
}
//============start_of_delete_face_elements================
/// Delete face elements and wipe the mesh
//==========================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
delete_face_elements(Mesh* const & boundary_mesh_pt)
{
 // How many surface elements are in the surface mesh
 unsigned n_element = boundary_mesh_pt->nelement();
 // Loop over the surface elements
 for(unsigned e=0;e<n_element;e++)
  {
   //   Kill surface element
   delete boundary_mesh_pt->element_pt(e);
  }

 // Wipe the mesh
 boundary_mesh_pt->flush_element_and_node_storage();

} // end of delete_face_elements
//============start_of_create_solid_traction_elements===================
/// Create solid traction elements
//=======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_solid_traction_elements()
```

```
{
 // Loop over pressure loaded boundaries
 unsigned b=0;
 unsigned nb=3;
 for (unsigned i=0;i<nb;i++)
  {
   switch(i)
    {
    case 0:
     b=Upper_inner_boundary_id;
     break;

    case 1:
     b=Lower_inner_boundary_id;
     break;

    case 2:
     b=Rib_divider_boundary_id;
     break;
    }

   // We're attaching face elements to region 0
   unsigned r=0;

   // How many bulk elements are adjacent to boundary b?
   unsigned n_element = Solid_mesh_pt->nboundary_element_in_region(b,r);

   // Loop over the bulk elements adjacent to boundary b
   for(unsigned e=0;e<n_element;e++)
    {
     // Get pointer to the bulk element that is adjacent to boundary b
     ELASTICITY_ELEMENT* bulk_elem_pt = dynamic_cast<ELASTICITY_ELEMENT*>(
      Solid_mesh_pt->boundary_element_in_region_pt(b,r,e));

     //Find the index of the face of element e along boundary b
     int face_index = Solid_mesh_pt->face_index_at_boundary_in_region(b,r,e);

     // Create element
     TimeHarmonicLinearElasticityTractionElement<ELASTICITY_ELEMENT>* el_pt=
      new TimeHarmonicLinearElasticityTractionElement<ELASTICITY_ELEMENT>
      (bulk_elem_pt,face_index);

     // Add to mesh
     Solid_traction_mesh_pt->add_element_pt(el_pt);

     // Associate element with bulk boundary (to allow it to access
     // the boundary coordinates in the bulk mesh)
     el_pt->set_boundary_number_in_bulk_mesh(b);

     //Set the traction function
     el_pt->traction_fct_pt() = Global_Parameters::pressure_load;
    }
  }
} // end of create_traction_elements
//============start_of_create_fsi_traction_elements======================
/// Create fsi traction elements
//=======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_fsi_traction_elements()
{
 // We're on the outer boundary of the solid mesh
 unsigned b=Outer_boundary_id;
 // How many bulk elements are adjacent to boundary b?
 unsigned n_element = Solid_mesh_pt->nboundary_element(b);

 // Loop over the bulk elements adjacent to boundary b
 for(unsigned e=0;e<n_element;e++)
  {
   // Get pointer to the bulk element that is adjacent to boundary b
   ELASTICITY_ELEMENT* bulk_elem_pt = dynamic_cast<ELASTICITY_ELEMENT*>(
    Solid_mesh_pt->boundary_element_pt(b,e));

   //Find the index of the face of element e along boundary b
   int face_index = Solid_mesh_pt->face_index_at_boundary(b,e);

   // Create element
   TimeHarmonicLinElastLoadedByHelmholtzPressureBCElement
    <ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>* el_pt=
    new TimeHarmonicLinElastLoadedByHelmholtzPressureBCElement
    <ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>(bulk_elem_pt,
                                           face_index);
   // Add to mesh
   FSI_traction_mesh_pt->add_element_pt(el_pt);

   // Associate element with bulk boundary (to allow it to access
   // the boundary coordinates in the bulk mesh)
```

```
   el_pt->set_boundary_number_in_bulk_mesh(b);

   // Set FSI parameter
   el_pt->q_pt()=&Global_Parameters::Q;
  }

} // end of create_traction_elements
//============start_of_create_helmholtz_fsi_flux_elements================
/// Create Helmholtz fsi flux elements
//=======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_helmholtz_fsi_flux_elements()
{

 // Attach to inner boundary of Helmholtz mesh (0)
 unsigned b=0;
 // How many bulk elements are adjacent to boundary b?
 unsigned n_element = Helmholtz_mesh_pt->nboundary_element(b);

 // Loop over the bulk elements adjacent to boundary b
 for(unsigned e=0;e<n_element;e++)
  {
   // Get pointer to the bulk element that is adjacent to boundary b
   HELMHOLTZ_ELEMENT* bulk_elem_pt = dynamic_cast<HELMHOLTZ_ELEMENT*>(
    Helmholtz_mesh_pt->boundary_element_pt(b,e));

   //Find the index of the face of element e along boundary b
   int face_index = Helmholtz_mesh_pt->face_index_at_boundary(b,e);

   // Create element
   HelmholtzFluxFromNormalDisplacementBCElement
    <HELMHOLTZ_ELEMENT,ELASTICITY_ELEMENT>* el_pt=
    new HelmholtzFluxFromNormalDisplacementBCElement
    <HELMHOLTZ_ELEMENT,ELASTICITY_ELEMENT>(bulk_elem_pt,
                                           face_index);

   // Add to mesh
   Helmholtz_fsi_flux_mesh_pt->add_element_pt(el_pt);

   // Associate element with bulk boundary (to allow it to access
   // the boundary coordinates in the bulk mesh)
   el_pt->set_boundary_number_in_bulk_mesh(b);
  }

} // end of create_helmholtz_flux_elements
//============start_of_create_ABC_elements==============================
/// Create ABC elements on the outer boundary of
/// the Helmholtz mesh
//===========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_helmholtz_ABC_elements()
{
 // We're on boundary 2 of the Helmholtz mesh
 unsigned b=2;
 // How many bulk elements are adjacent to boundary b?
 unsigned n_element = Helmholtz_mesh_pt->nboundary_element(b);

 // Loop over the bulk elements adjacent to boundary b?
 for(unsigned e=0;e<n_element;e++)
  {
   // Get pointer to the bulk element that is adjacent to boundary b
   HELMHOLTZ_ELEMENT* bulk_elem_pt = dynamic_cast<HELMHOLTZ_ELEMENT*>(
    Helmholtz_mesh_pt->boundary_element_pt(b,e));

   //Find the index of the face of element e along boundary b
   int face_index = Helmholtz_mesh_pt->face_index_at_boundary(b,e);

   // Build the corresponding ABC element
   HelmholtzAbsorbingBCElement<HELMHOLTZ_ELEMENT>* flux_element_pt = new
    HelmholtzAbsorbingBCElement<HELMHOLTZ_ELEMENT>(bulk_elem_pt,face_index);
   // Set pointer to outer radius of artificial boundary
   flux_element_pt->outer_radius_pt()=&Global_Parameters::Outer_radius;

   // Set order of absorbing boundary condition
   flux_element_pt->abc_order_pt()=&Global_Parameters::ABC_order;
   //Add the flux boundary element to the  helmholtz_outer_boundary_mesh
   Helmholtz_outer_boundary_mesh_pt->add_element_pt(flux_element_pt);
  }
} // end of create_helmholtz_ABC_elements
//=====================start_of_setup_interaction======================
/// Setup interaction between two fields
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
setup_interaction()
```

```
{
 // Setup Helmholtz "pressure" load on traction elements
 unsigned boundary_in_helmholtz_mesh=0;
 // Doc boundary coordinate for Helmholtz
 ofstream the_file;
 the_file.open("boundary_coordinate_hh.dat");
 Helmholtz_mesh_pt->Mesh::doc_boundary_coordinates<HELMHOLTZ_ELEMENT>
  (boundary_in_helmholtz_mesh, the_file);
 the_file.close();
  Multi_domain_functions::setup_bulk_elements_adjacent_to_face_mesh
  <HELMHOLTZ_ELEMENT,2>
  (this,boundary_in_helmholtz_mesh,Helmholtz_mesh_pt,FSI_traction_mesh_pt);
 // Setup Helmholtz flux from normal displacement interaction
 unsigned boundary_in_solid_mesh=Outer_boundary_id;
 // Doc boundary coordinate for solid mesh
 the_file.open("boundary_coordinate_solid.dat");
 Solid_mesh_pt->Mesh::template doc_boundary_coordinates<ELASTICITY_ELEMENT>
  (boundary_in_solid_mesh, the_file);
 the_file.close();
 Multi_domain_functions::setup_bulk_elements_adjacent_to_face_mesh
  <ELASTICITY_ELEMENT,2>(
   this,boundary_in_solid_mesh,Solid_mesh_pt,Helmholtz_fsi_flux_mesh_pt);
}
//==============start_doc===========================================
/// Doc the solution
//==================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedDiskProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::doc_solution()
{
 ofstream some_file,some_file2;
 char filename[100];
 // Number of plot points
 unsigned n_plot=5;
 // Compute/output the radiated power
 //----------------------------------
 sprintf(filename,"%s/power%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 // Accumulate contribution from elements
 double power=0.0;
 unsigned nn_element=Helmholtz_outer_boundary_mesh_pt->nelement();
 for(unsigned e=0;e<nn_element;e++)
  {
   HelmholtzBCElementBase<HELMHOLTZ_ELEMENT> *el_pt =
    dynamic_cast<HelmholtzBCElementBase<HELMHOLTZ_ELEMENT>*>(
     Helmholtz_outer_boundary_mesh_pt->element_pt(e));
   power += el_pt->global_power_contribution(some_file);
  }
 some_file.close();
 oomph_info « "Step: " « Doc_info.number()
            « ": Q=" « Global_Parameters::Q  « "\n"
            « " k_squared=" « Global_Parameters::K_squared  « "\n"
            « " density ratio (annulus) ="
            « Global_Parameters::Density_ratio[0]  « "\n"
            « " density ratio (rib)     ="
            « Global_Parameters::Density_ratio[1]  « "\n"
            « " omega_sq (annulus)=" « Global_Parameters::Omega_sq[0]  « "\n"
            « " omega_sq (rib    )=" « Global_Parameters::Omega_sq[1]  « "\n"
            « " Total radiated power " « power  « "\n"
            « std::endl;
 // Write trace file
 Trace_file « Global_Parameters::Q « " "
            « Global_Parameters::K_squared « " "
            « Global_Parameters::Density_ratio[0] « " "
            « Global_Parameters::Density_ratio[1] « " "
            « Global_Parameters::Omega_sq[0] « " "
            « Global_Parameters::Omega_sq[1] « " "
            « power « " "
            « std::endl;

 std::ostringstream case_string;
 case_string « "TEXT X=10,Y=90, T=\"Q="
             «  Global_Parameters::Q
             « ",  k<sup>2</sup> = "
             «  Global_Parameters::K_squared
             « ",  density ratio = "
             «  Global_Parameters::Density_ratio[0] « " and "
             «  Global_Parameters::Density_ratio[1]
             « ",  omega_sq = "
             «  Global_Parameters::Omega_sq[0] « " and "
             «  Global_Parameters::Omega_sq[1] « " "
             « "\"\n";
 // Output displacement field
 //--------------------------
 sprintf(filename,"%s/elast_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
```

```
 Solid_mesh_pt->output(some_file,n_plot);
 some_file.close();
 // Output solid traction elements
 //--------------------------------
 sprintf(filename,"%s/solid_traction_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Solid_traction_mesh_pt->output(some_file,n_plot);
 some_file.close();
 // Output fsi traction elements
 //-----------------------------
 sprintf(filename,"%s/traction_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 FSI_traction_mesh_pt->output(some_file,n_plot);
 some_file.close();
 // Output Helmholtz fsi flux elements
 //-----------------------------------
 sprintf(filename,"%s/flux_bc_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Helmholtz_fsi_flux_mesh_pt->output(some_file,n_plot);
 some_file.close();
 // Output Helmholtz
 //-----------------
 sprintf(filename,"%s/helmholtz_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Helmholtz_mesh_pt->output(some_file,n_plot);
 some_file « case_string.str();
 some_file.close();

 // Output regions of solid mesh
 //-----------------------------
 unsigned nreg=Solid_mesh_pt->nregion();
 for (unsigned r=0;r<nreg;r++)
  {
   sprintf(filename,"%s/region%i_%i.dat",Doc_info.directory().c_str(),
           r,Doc_info.number());
   some_file.open(filename);
   unsigned nel=Solid_mesh_pt->nregion_element(r);
   for (unsigned e=0;e<nel;e++)
    {
     FiniteElement* el_pt=Solid_mesh_pt->region_element_pt(r,e);
     el_pt->output(some_file,n_plot);
    }
   some_file.close();
  }


 // Do animation of Helmholtz solution
 //-----------------------------------
 unsigned nstep=40;
 for (unsigned i=0;i<nstep;i++)
  {
   sprintf(filename,"%s/helmholtz_animation%i_frame%i.dat",
           Doc_info.directory().c_str(),
           Doc_info.number(),i);
   some_file.open(filename);
   double phi=2.0*MathematicalConstants::Pi*double(i)/double(nstep-1);
   unsigned nelem=Helmholtz_mesh_pt->nelement();
   for (unsigned e=0;e<nelem;e++)
    {
     HELMHOLTZ_ELEMENT* el_pt=dynamic_cast<HELMHOLTZ_ELEMENT*>(
      Helmholtz_mesh_pt->element_pt(e));
     el_pt->output_real(some_file,phi,n_plot);
    }
   some_file.close();
  }
 cout « "Doced for Q=" « Global_Parameters::Q « " (step "
      « Doc_info.number() « ")\n";

// Increment label for output files
 Doc_info.number()++;

} //end doc
//=======start_of_main==================================================
/// Driver for acoustic fsi problem
//======================================================================
int main(int argc, char **argv)
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);
 // Define possible command line arguments and parse the ones that
 // were actually specified

 // Output directory
```

```
 CommandLineArgs::specify_command_line_flag("--dir",
                                            &Global_Parameters::Directory);

 // Peakiness parameter for loading
 CommandLineArgs::specify_command_line_flag("--alpha",
                                            &Global_Parameters::Alpha);
 // Multiplier for number of elements in solid mesh
 CommandLineArgs::specify_command_line_flag("--el_multiplier",
                            &Global_Parameters::El_multiplier);

 // Outer radius of Helmholtz domain
 CommandLineArgs::specify_command_line_flag("--outer_radius",
                            &Global_Parameters::Outer_radius);

 // Validaton run?
 CommandLineArgs::specify_command_line_flag("--validation");

 // Max. number of adaptations
 unsigned max_adapt=3;
 CommandLineArgs::specify_command_line_flag("--max_adapt",&max_adapt);

 // Parse command line
 CommandLineArgs::parse_and_assign();

 // Doc what has actually been specified on the command line
 CommandLineArgs::doc_specified_flags();
 //Set up the problem
 CoatedDiskProblem<ProjectableTimeHarmonicLinearElasticityElement
                   <TTimeHarmonicLinearElasticityElement<2,3> >,
                   RefineableQHelmholtzElement<2,3> > problem;
 // Set values for parameter values
 Global_Parameters::Q=5.0;
 Global_Parameters::Density_ratio[0]=0.1;
 Global_Parameters::Density_ratio[1]=0.1;
 Global_Parameters::update_parameter_values();

 //Parameter study
 unsigned nstep=3;
 if (CommandLineArgs::command_line_flag_has_been_set("--validation"))
  {
   nstep=1;
   max_adapt=2;
  }
 for(unsigned i=0;i<nstep;i++)
  {
   // Solve the problem with Newton's method, allowing
   // up to max_adapt mesh adaptations after every solve.
   problem.newton_solve(max_adapt);
   // Doc solution
   problem.doc_solution();
   // Make rib a lot heavier but keep its stiffness
   if (i==0)
    {
     Global_Parameters::E_pt[1]->update_constitutive_parameters(
      Global_Parameters::Nu,1.0);
     Global_Parameters::Density_ratio[1]=10.0;
     Global_Parameters::update_parameter_values();
    }
   // Make rib very soft and inertia-less
   if (i==1)
    {
     Global_Parameters::E_pt[1]->update_constitutive_parameters(
      Global_Parameters::Nu,1.0e-16);
     Global_Parameters::Density_ratio[1]=0.0;
     Global_Parameters::update_parameter_values();
    }


  }

} //end of main
```

1.5 Source files for this tutorial

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

  demo_drivers/interaction/acoustic_fsi/

- The driver code is:

`demo_drivers/interaction/acoustic_fsi/unstructured_acoustic_fsi.cc`

1.6 PDF file

A `pdf version` of this document is available.