Chapter 1

The spatially-adaptive solution of azimuthally Fourier-decomposed time-harmonic 3D acoustic fluid-structure interaction problems on unstructured meshes

In this document we discuss the spatially-adaptive solution of 3D time-harmonic acoustic fluid-structure interaction problems in cylindrical polar coordinates 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 adaptive, unstructured meshes and the assignment of different material properties to different parts of the solid domain.

1.1 A test problem

The sketch below shows the problem setup: An elastic sphere which is reinforced with an azimuthal T-rib is immersed in a compressible fluid and subjected to a time-periodic pressure load of magnitude

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

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

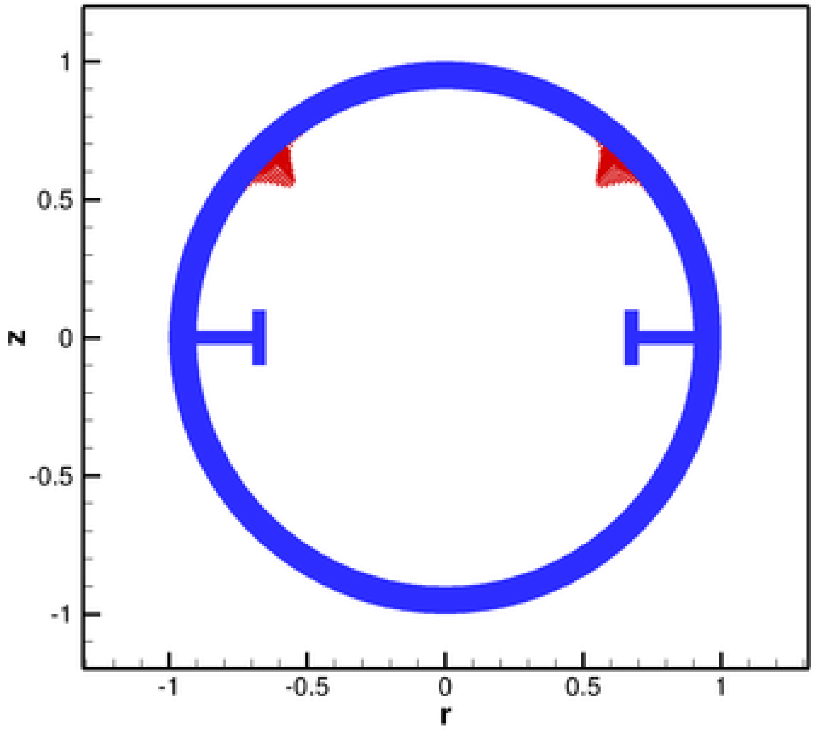

Figure 1.1 Sketch of the problem setup.

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 undeformed configuration. The left half of the plot is used to show the (mirror image of the) adaptive unstructured mesh on which the displacement field was computed:

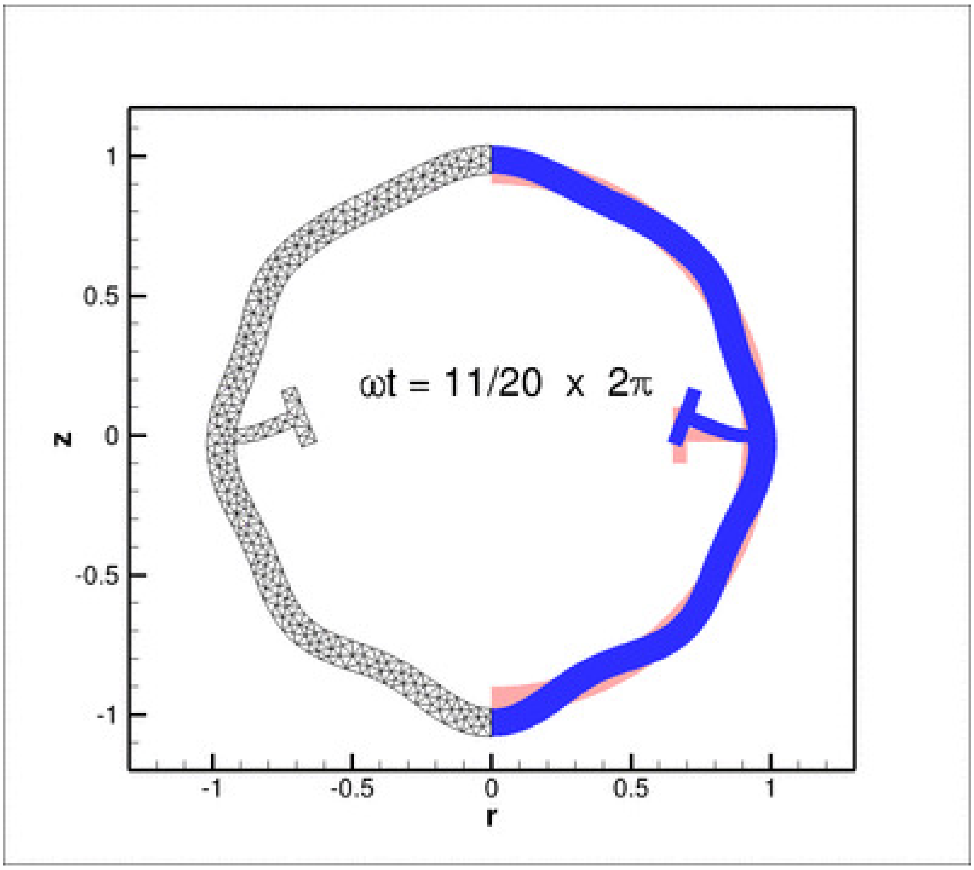

Figure 1.2 Animation of the time-harmonic deformation.

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

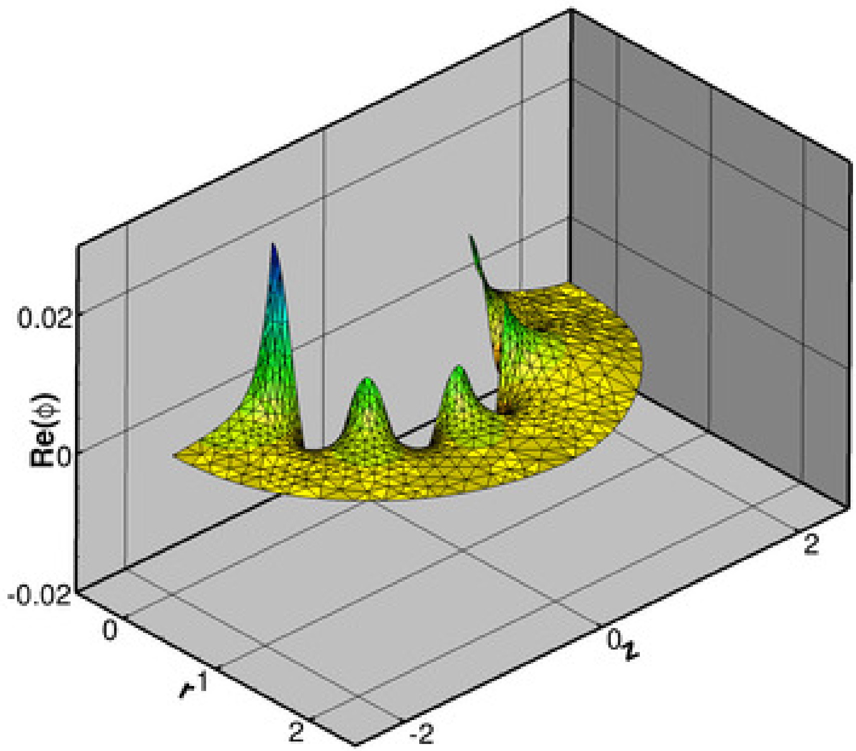

Figure 1.3 The fluid displacement potential, a measure of the fluid pressure. Elevation: real part; contours: imaginary part.

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 sphere and set its inertia to zero, the rib will not offer much structural resistance and the sphere will deform as if the rib was not present. If we then set $\alpha = 0$ we apply a spherically symmetric forcing onto the structure and would expect the resulting displacement field (at least in the sphere) to be spherically symmetric, too.
The animation of the displacement field for this case, shown below, shows that this is indeed the case:

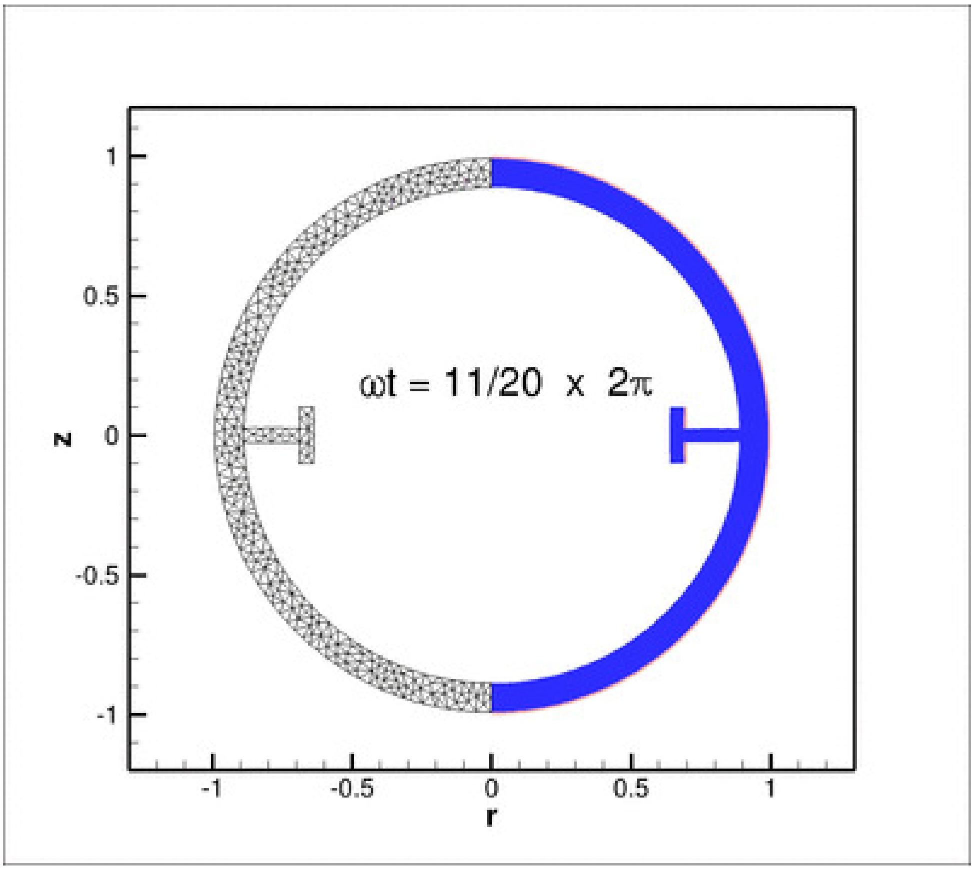

Figure 1.4 Animation of the time-harmonic deformation for a uniform pressure load and a very soft and inertia-less rib.

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

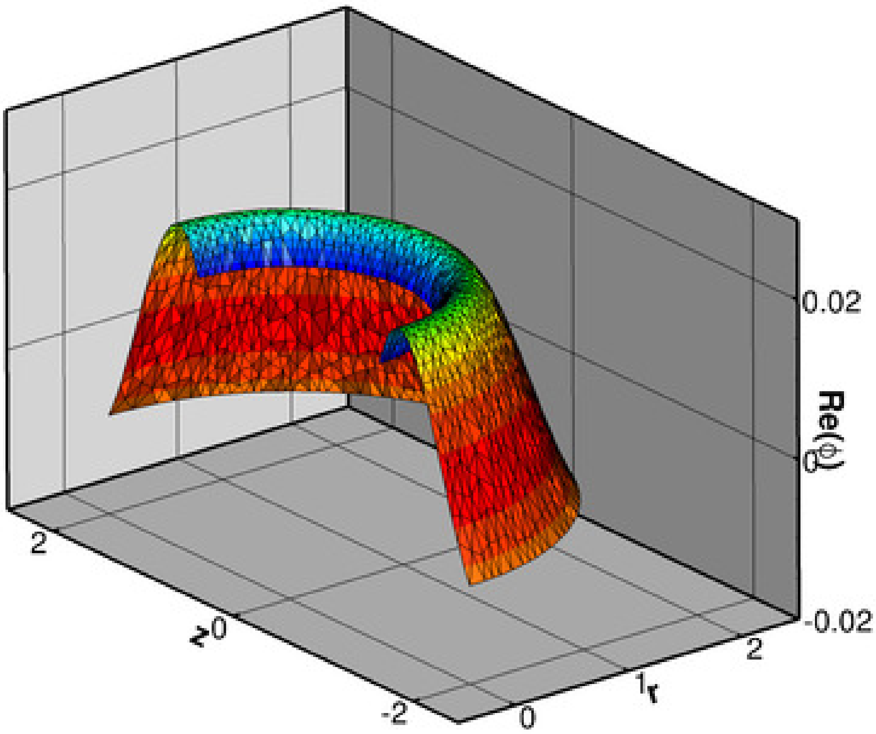

Figure 1.5 The fluid displacement potential, a measure of the fluid pressure for a uniform pressure load and a very soft and inertia-less rib. Elevation: real part; contours: imaginary part.

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/fourier_decomposed_acoustic_fsi/fourier_decomposed_acoustic_fsi.cc`

and

`demo_drivers/interaction/fourier_decomposed_acoustic_fsi/unstructured_fourier_decomposed_acoustic_fsi.cc`

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

- The provision of multiple non-dimensional Young's moduli and frequency parameters for the two different regions (the rib and the sphere). Recall that the non-dimensional Young's modulus specified via `TimeHarmonicFourierDecomposedLinearElasticityEquations::youngs_modulus_pt()` represents the ratio of the material's actual Young's modulus to the Young's modulus used in the non-dimensionalisation of the equations.

- 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 the `actions_before/after_adapt()` functions and 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 sphere) 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// Copyright (C) 2006-2023 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
#include <complex>
#include <cmath>

//Oomph-lib includes
#include "generic.h"

//The Helmholtz equation
#include "fourier_decomposed_helmholtz.h"

//The Elasticity equation
#include "time_harmonic_fourier_decomposed_linear_elasticity.h"

// The interaction elements
#include "multi_physics.h"

// The meshes
#include "meshes/annular_mesh.h"
#include "meshes/triangle_mesh.h"

// Get the Bessel functions
#include "oomph_crbond_bessel.h"

using namespace oomph;
using namespace std;


/// ////////////////////////////////////////////////////////////////////
/// ////////////////////////////////////////////////////////////////////
// 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");
  }

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

 /// 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_of_namespace==========================================
/// Global variables
//===================================================================
namespace Global_Parameters
{

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

 /// Radius of outer boundary of Helmholtz domain
 double Outer_radius=2.0;

 /// FSI parameter
 double Q=10.0;

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

 /// Define azimuthal Fourier wavenumber
 int Fourier_wavenumber=0;

 /// Poisson's ratio Nu
 std::complex<double> Nu(std::complex<double>(0.3,0.0));

 /// Define the non-dimensional Young's modulus
 Vector<std::complex<double> > E(2,std::complex<double>(1.0,0.0));

 /// Non-dim square of frequency for solid -- dependent variable!
 Vector<std::complex<double> > Omega_sq(2,std::complex<double>(100.0,0.0));

 /// Density ratio: solid to fluid
Vector<double> Density_ratio(2,1.0);

 /// 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=0.0;

 /// 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 = 2;
  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";
} //end_of_namespace



//=============start_of_problem_class===================================
/// Coated sphere FSI
//======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
class CoatedSphereProblem : public Problem
{

public:

 /// Constructor:
 CoatedSphereProblem();

 /// 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();

 /// Recompute gamma integral before checking Newton residuals
 void actions_before_newton_convergence_check()
  {
   Helmholtz_DtN_mesh_pt->setup_gamma();
  }

 /// Doc the solution
 void doc_solution(DocInfo& doc_info);

private:

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

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

 /// Setup interaction
 void setup_interaction();

 /// Create DtN elements on outer boundary
 void create_helmholtz_DtN_elements();

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

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

 /// 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;

 /// Boundary ID of outer boundary in Helmholtz mesh
 unsigned HH_outer_boundary_id;

 /// Boundary ID of inner boundary in Helmholtz mesh
 unsigned HH_inner_boundary_id;

 /// Boundary ID of upper boundary in Helmholtz mesh
 unsigned HH_upper_symmetry_boundary_id;

 /// Boundary ID of lower boundary in Helmholtz mesh
 unsigned HH_lower_symmetry_boundary_id;

#ifdef ADAPTIVE

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

#else

 /// Pointer to solid mesh
 TriangleMesh<ELASTICITY_ELEMENT>* Solid_mesh_pt;

#endif


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

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

#ifdef ADAPTIVE

 /// Pointer to Helmholtz mesh
 RefineableTriangleMesh<HELMHOLTZ_ELEMENT>* Helmholtz_mesh_pt;

#else

 /// Pointer to Helmholtz mesh
 TriangleMesh<HELMHOLTZ_ELEMENT>* Helmholtz_mesh_pt;

#endif

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

 /// Pointer to mesh containing the DtN elements
 FourierDecomposedHelmholtzDtNMesh<HELMHOLTZ_ELEMENT>* Helmholtz_DtN_mesh_pt;

 /// Trace file
 ofstream Trace_file;

};// end_of_problem_class


//===========start_of_constructor=======================================
/// Constructor:
//======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
CoatedSphereProblem<ELASTICITY_ELEMENT, HELMHOLTZ_ELEMENT>::
CoatedSphereProblem()
{

 // 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);
#ifdef ADAPTIVE

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

#else

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

#endif

 }
 // Helmholtz mesh
 //---------------
 {
  // Start and end coordinates
  Vector<double> r_start(2);
  Vector<double> r_end(2);
  // Inner radius of helmholtz region
  double r_inner = 1.0;

  // Outer radius of Helmholtz region
  double r_outer = Global_Parameters::Outer_radius;

  // 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
  HH_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
  HH_upper_symmetry_boundary_id=boundary_id;
  // Inner boundary
  //---------------
  Ellipse* upper_inner_boundary_pt =
   new Ellipse(r_inner,r_inner);
  zeta_start=0.5*MathematicalConstants::Pi;
  zeta_end=-0.5*MathematicalConstants::Pi;
  nsegment=40;
  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
  HH_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
  HH_lower_symmetry_boundary_id=boundary_id;

  // Combine to curvilinear boundary
  //--------------------------------
  closed_curve_pt=
   new TriangleMeshClosedCurve(curvilinear_boundary_pt);

  // 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;
#ifdef ADAPTIVE

  // Build the mesh
  Helmholtz_mesh_pt=new
   RefineableTriangleMesh<HELMHOLTZ_ELEMENT>(triangle_mesh_parameters);

#else

  // Build the mesh
  Helmholtz_mesh_pt=new
   TriangleMesh<HELMHOLTZ_ELEMENT>(triangle_mesh_parameters);

#endif

 }

 // Create mesh for DtN elements on outer boundary
 unsigned nfourier=20;
 Helmholtz_DtN_mesh_pt=
  new FourierDecomposedHelmholtzDtNMesh<HELMHOLTZ_ELEMENT>(
   Global_Parameters::Outer_radius,nfourier);


#ifdef ADAPTIVE

 // Set error estimators
 Solid_mesh_pt->spatial_error_estimator_pt()=new Z2ErrorEstimator;
 Helmholtz_mesh_pt->spatial_error_estimator_pt()=new Z2ErrorEstimator;

#endif

 // 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 DtN elements
 create_helmholtz_DtN_elements();

 // Combine sub meshes
 //-------------------
 add_sub_mesh(Solid_mesh_pt);
 add_sub_mesh(Solid_traction_mesh_pt);
 add_sub_mesh(FSI_traction_mesh_pt);
 add_sub_mesh(Helmholtz_mesh_pt);
 add_sub_mesh(Helmholtz_fsi_flux_mesh_pt);
 add_sub_mesh(Helmholtz_DtN_mesh_pt);
 // Build the Problem's global mesh from its various sub-meshes
 build_global_mesh();
 // Complete problem setup
 complete_problem_setup();
 // Setup fluid-structure interaction
 //----------------------------------
 setup_interaction();

 // Open trace file
 char filename[100];
 sprintf(filename,"%s/trace.dat",Global_Parameters::Directory.c_str());
 Trace_file.open(filename);

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

}//end_of_constructor


//====================start_of_actions_before_adapt=====================
/// Actions before adapt: Wipe the meshes face elements
//======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<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_DtN_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 CoatedSphereProblem<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 DtN elements from all elements that are
 // adjacent to the outer boundary of Helmholtz mesh
 create_helmholtz_DtN_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_actions_after_adapt======================
/// Complete problem setup: Apply boundary conditions and set
/// physical properties
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
complete_problem_setup()
{

 // Solid boundary conditions:
 //---------------------------
 // Pin real and imag part of horizontal and azimuthal 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 r-displacement
    nod_pt->pin(0);
    nod_pt->set_value(0,0.0);

    // Imag part of r-displacement
    nod_pt->pin(3);
    nod_pt->set_value(3,0.0);

    // Real part of phi-displacement
    nod_pt->pin(2);
    nod_pt->set_value(2,0.0);

    // Imag part of phi-displacement
    nod_pt->pin(5);
    nod_pt->set_value(5,0.0);
   }
 }
 {
  //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 r-displacement
    nod_pt->pin(0);
    nod_pt->set_value(0,0.0);

    // Imag part of r-displacement
    nod_pt->pin(3);
    nod_pt->set_value(3,0.0);

    // Real part of phi-displacement
    nod_pt->pin(2);
    nod_pt->set_value(2,0.0);

    // Imag part of phi-displacement
    nod_pt->pin(5);
    nod_pt->set_value(5,0.0);
   }
 }


 //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 pointer to Fourier wavenumber
     el_pt->fourier_wavenumber_pt() = &Global_Parameters::Fourier_wavenumber;
```

```
     // Set the pointer to Poisson's ratio
     el_pt->nu_pt() = &Global_Parameters::Nu;

     // Square of non-dim frequency
     el_pt->omega_sq_pt()= &Global_Parameters::Omega_sq[r];

     // Set the pointer to non-dim Young's modulus
     el_pt->youngs_modulus_pt() = &Global_Parameters::E[r];

    }
  }

 // Complete the build of all Helmholtz elements so they are fully functional
 unsigned n_element = Helmholtz_mesh_pt->nelement();
 for(unsigned i=0;i<n_element;i++)
  {
   // Upcast from GeneralsedElement to the present element
   HELMHOLTZ_ELEMENT *el_pt = dynamic_cast<HELMHOLTZ_ELEMENT*>(
    Helmholtz_mesh_pt->element_pt(i));

   //Set the k_squared pointer
   el_pt->k_squared_pt()=&Global_Parameters::K_squared;

   // Set pointer to Fourier wave number
   el_pt->fourier_wavenumber_pt()=&Global_Parameters::Fourier_wavenumber;
  }

}



//============start_of_delete_face_elements================
/// Delete face elements and wipe the mesh
//==========================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<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_outer_bc_elements===========================
/// Create BC elements on outer boundary
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_helmholtz_DtN_elements()
{
 // Outer boundary:
 unsigned b=HH_outer_boundary_id;

 // Loop over the bulk elements adjacent to boundary b?
 unsigned n_element = Helmholtz_mesh_pt->nboundary_element(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 DtN element
   FourierDecomposedHelmholtzDtNBoundaryElement<HELMHOLTZ_ELEMENT>*
    flux_element_pt = new
    FourierDecomposedHelmholtzDtNBoundaryElement<HELMHOLTZ_ELEMENT>
    (bulk_elem_pt,face_index);

   //Add the flux boundary element to the  helmholtz_DtN_mesh
   Helmholtz_DtN_mesh_pt->add_element_pt(flux_element_pt);

   // Set pointer to the mesh that contains all the boundary condition
   // elements on this boundary
   flux_element_pt->set_outer_boundary_mesh_pt(Helmholtz_DtN_mesh_pt);
```

```
  }
} // end_of_create_outer_bc_elements




//=====================start_of_setup_interaction======================
/// Setup interaction between two fields
//========================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
setup_interaction()
{

 // Setup Helmholtz "pressure" load on traction elements
 unsigned boundary_in_helmholtz_mesh=HH_inner_boundary_id;

 // Doc boundary coordinate for Helmholtz
 ofstream the_file;
 the_file.open("boundary_coordinate_hh.dat");
 Helmholtz_mesh_pt->Mesh::template doc_boundary_coordinates<HELMHOLTZ_ELEMENT>
  (boundary_in_helmholtz_mesh, the_file);
 the_file.close();

 // Setup interaction
  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();

 // Setup interaction
 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);

}// end_of_setup_interaction




//============start_of_create_fsi_traction_elements======================
/// Create fsi traction elements
//=======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_fsi_traction_elements()
{
 // We're on 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
   FourierDecomposedTimeHarmonicLinElastLoadedByHelmholtzPressureBCElement
    <ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>* el_pt=
    new FourierDecomposedTimeHarmonicLinElastLoadedByHelmholtzPressureBCElement
    <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_fsi_traction_elements



//============start_of_create_helmholtz_fsi_flux_elements================
/// Create Helmholtz fsi flux elements
//=======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
create_helmholtz_fsi_flux_elements()
{
 // Attach to inner boundary of Helmholtz mesh
 unsigned b=HH_inner_boundary_id;

 // 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
   FourierDecomposedHelmholtzFluxFromNormalDisplacementBCElement
    <HELMHOLTZ_ELEMENT,ELASTICITY_ELEMENT>* el_pt=
    new FourierDecomposedHelmholtzFluxFromNormalDisplacementBCElement
    <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_fsi_flux_elements


//============start_of_create_solid_traction_elements====================
/// Create solid traction elements
//=======================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<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
     TimeHarmonicFourierDecomposedLinearElasticityTractionElement
      <ELASTICITY_ELEMENT>* el_pt=
      new TimeHarmonicFourierDecomposedLinearElasticityTractionElement
      <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_doc_solution==============================
/// Doc the solution
//=================================================================
template<class ELASTICITY_ELEMENT, class HELMHOLTZ_ELEMENT>
void CoatedSphereProblem<ELASTICITY_ELEMENT,HELMHOLTZ_ELEMENT>::
doc_solution(DocInfo& doc_info)
{

 // Doc parameters
 oomph_info « "Writing result for step " « doc_info.number()
            « ". Parameters: "« std::endl;
 oomph_info « "Fourier mode number : N = "
            « Global_Parameters::Fourier_wavenumber « std::endl;
 oomph_info « "FSI parameter : Q = " « Global_Parameters::Q « std::endl;
 oomph_info « "Fluid outer radius : R = " « Global_Parameters::Outer_radius
            « std::endl;
 oomph_info « "Fluid wavenumber : k^2 = " « Global_Parameters::K_squared
            « std::endl;
 oomph_info « "Solid wavenumber : Omega_sq = "
            « Global_Parameters::Omega_sq[0] « std::endl;
 oomph_info « "Solid wavenumber : Omega_sq = "
            « Global_Parameters::Omega_sq[1]
            « std::endl « std::endl;


 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_DtN_mesh_pt->nelement();
 for(unsigned e=0;e<nn_element;e++)
  {
   FourierDecomposedHelmholtzBCElementBase<HELMHOLTZ_ELEMENT> *el_pt =
    dynamic_cast<FourierDecomposedHelmholtzBCElementBase<HELMHOLTZ_ELEMENT>*>(
     Helmholtz_DtN_mesh_pt->element_pt(e));
   power += el_pt->global_power_contribution(some_file);
  }
 some_file.close();
 oomph_info « "Radiated power: " « power « std::endl;

 // 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 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.close();


 // Output fsi traction elements
 //----------------------------
 sprintf(filename,"%s/fsi_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/fsi_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();

 // 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].real() « " "
            « Global_Parameters::Omega_sq[1].real() « " "
            « power « " "
            « std::endl;

 // Bump up counter
 doc_info.number()++;

} //end_of_doc_solution



//=======start_of_main==================================================
/// Driver for coated sphere loaded by lineared fluid loading
//======================================================================
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);
 // Parameter controlling the sharpness of the pressure load
 CommandLineArgs::specify_command_line_flag("--alpha",
                                            &Global_Parameters::Alpha);
 // Parse command line
 CommandLineArgs::parse_and_assign();
 // Doc what has actually been specified on the command line
 CommandLineArgs::doc_specified_flags();



 // 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();

 // Update dependent parameters values
 Global_Parameters::update_parameter_values();

 // Set up doc info
 DocInfo doc_info;
 // Set output directory
 doc_info.set_directory(Global_Parameters::Directory);
#ifdef ADAPTIVE

 // Set up the problem
 CoatedSphereProblem<
  ProjectableTimeHarmonicFourierDecomposedLinearElasticityElement
  <TTimeHarmonicFourierDecomposedLinearElasticityElement<3> >,
  ProjectableFourierDecomposedHelmholtzElement
  <TFourierDecomposedHelmholtzElement<3> > > problem;

#else

 // Set up the problem
 CoatedSphereProblem<TTimeHarmonicFourierDecomposedLinearElasticityElement<3>,
```

```
                     TFourierDecomposedHelmholtzElement<3> > problem;
#endif

 //Parameter incrementation
 unsigned nstep=3;
 for(unsigned i=0;i<nstep;i++)
  {
#ifdef ADAPTIVE
   // Solve the problem with Newton's method, allowing
   // up to max_adapt mesh adaptations after every solve.
   unsigned max_adapt=3;
   problem.newton_solve(max_adapt);
#else
   // Solve the problem with Newton's method
   problem.newton_solve();
#endif
   // Doc solution
   problem.doc_solution(doc_info);

   // Make rib a lot heavier but keep its stiffness
   if (i==0)
    {
     Global_Parameters::E[1]=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[1]=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/fourier_decomposed_acoustic_fsi/

- The driver code is:

  demo_drivers/interaction/fourier_decomposed_acoustic_fsi/unstructured↩
  _fourier_decomposed_acoustic_fsi.cc

1.6 PDF file

A pdf version of this document is available.