Chapter 1

The adaptive solution of the equations of time-harmonic linear elasticity on unstructured meshes

The aim of this tutorial is to demonstrate the adaptive solution of the time-harmonic equations of linear elasticity in cartesian 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 unstructured mesh and the assignment of different material properties to different parts of the domain.

1.1 A test problem: Time-harmonic oscillations of an elastic annulus reinforced by a T-rib

The sketch below shows the problem setup: A 2D elastic annulus which is reinforced with two T-ribs is 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 outer edge. 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 $\alpha = 1$.

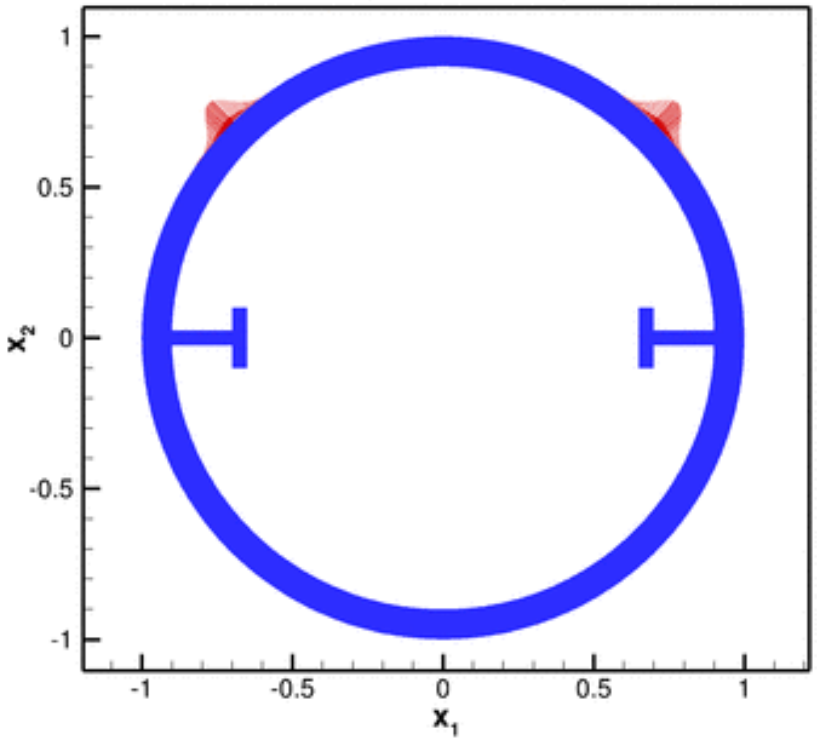

Figure 1.1 Sketch of the problem setup.

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:

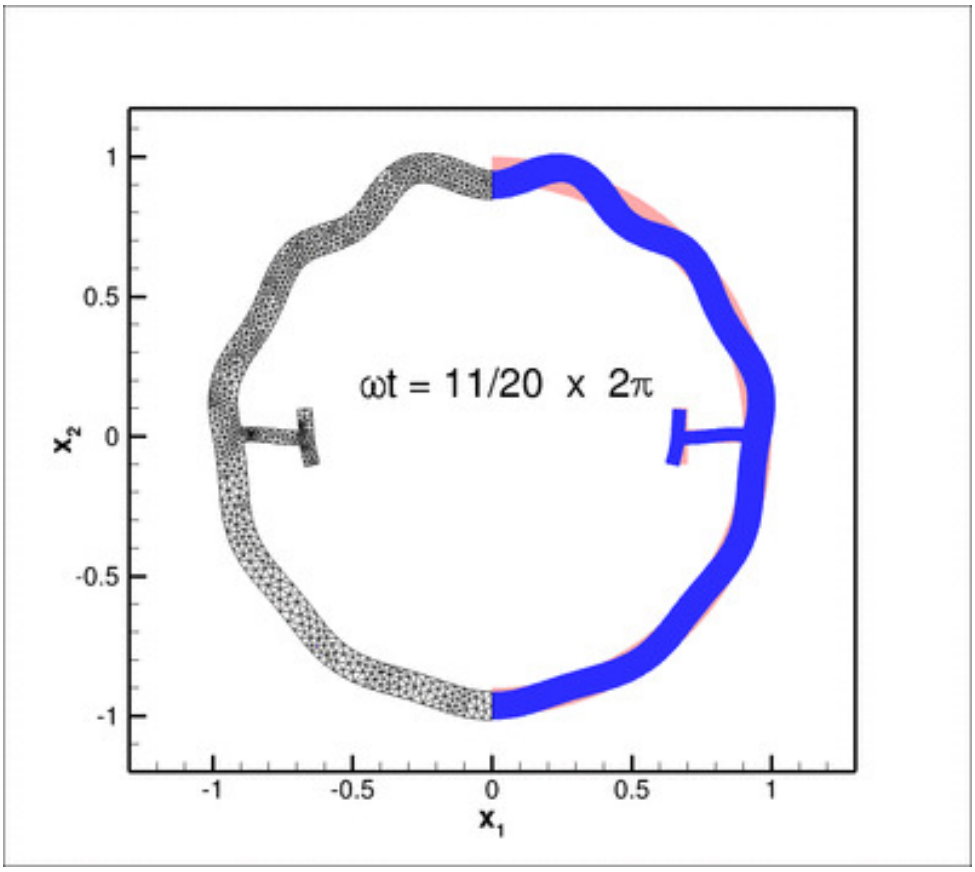

Figure 1.2 Animation of the time-harmonic deformation.

This looks very pretty and shows that we can compute 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, 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. For this case it is easy to find an analytical solution to the problem. The next two figures show a comparison between the analytical (green spheres) and computed solutions (shaded) for the real part of the horizontal and vertical displacements, respectively.

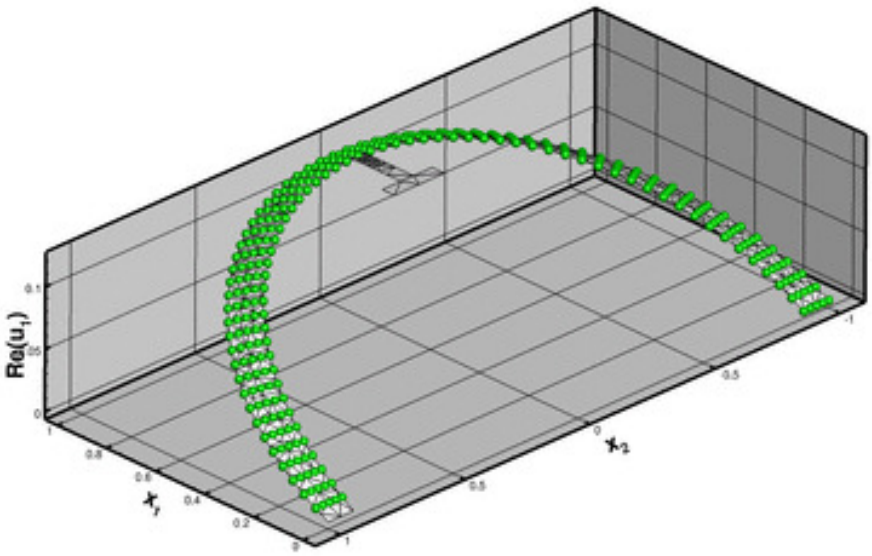

Figure 1.3 Real part of the horizontal displacement, computed (shaded) and exact (spheres).

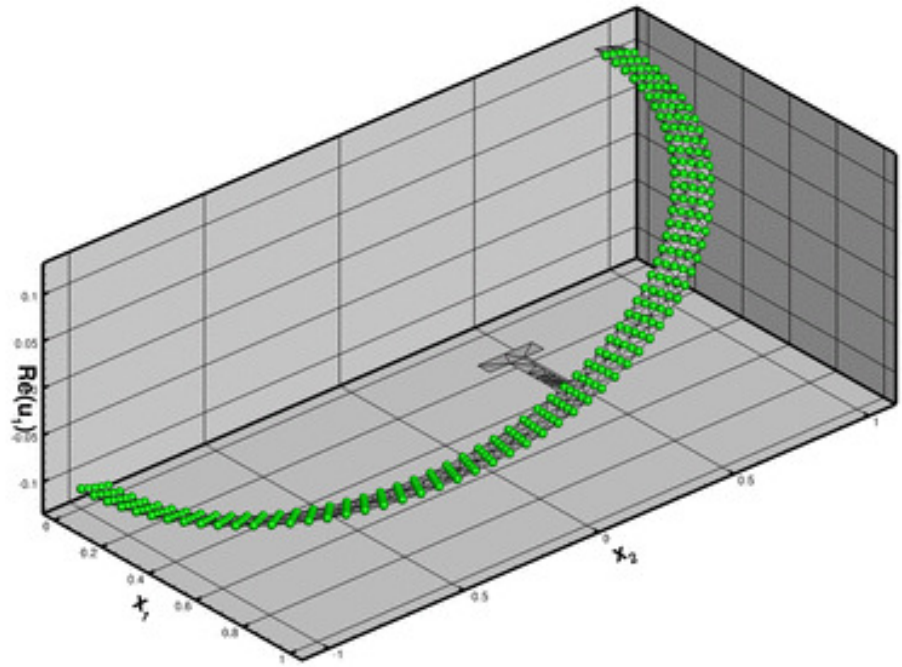

Figure 1.4 Real part of the vertical displacement, computed (shaded) and exact (spheres).

Convinced?

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/time_harmonic_linear_elasticity/elastic_annulus/time_↩
harmonic_elastic_annulus.cc
```

and

```
demo_drivers/time_harmonic_linear_elasticity/elastic_annulus/unstructured↩
_time_harmonic_elastic_annulus.cc
```

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

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

- 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 the mesh 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) 2006-2021 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//====================================================================
// Time-harmonic deformation of an T-rib reinforced elastic annulus subject to
// a nonuniform pressure load.
//Oomph-lib includes
#include "generic.h"
#include "time_harmonic_linear_elasticity.h"
//The meshes
#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
{

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

 /// Square of non-dim frequency
 double Omega_sq=10.0;

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

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

 /// Thickness of annulus
 double H_annulus=0.1;

 /// Uniform pressure
 double P = 0.1;

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

 /// Constant 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";

} //end namespace
//=============begin_problem============================================
/// Annular disk
//======================================================================
template<class ELASTICITY_ELEMENT>
class RingWithTRibProblem : public Problem
{
public:

 /// Constructor:
 RingWithTRibProblem();

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

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

 /// Actions before adapt: Wipe the mesh of traction elements
 void actions_before_adapt();
```

```
 /// Actions after adapt: Rebuild the mesh of traction elements
 void actions_after_adapt();

 /// Doc the solution
 void doc_solution();
private:

 /// \short Create traction elements
 void create_traction_elements();

 /// Delete traction elements
 void delete_traction_elements();

 /// Helper function to complete problem setup
 void complete_problem_setup();
#ifdef ADAPTIVE

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

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

 /// Pointer to mesh of traction elements
 Mesh* Traction_mesh_pt;

 /// DocInfo object for output
 DocInfo Doc_info;

 /// 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 outer boundary
 unsigned Outer_boundary_id;

};
//===========start_of_constructor=======================================
/// Constructor:
//======================================================================
template<class ELASTICITY_ELEMENT>
RingWithTRibProblem<ELASTICITY_ELEMENT>::RingWithTRibProblem()
{

 // 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_annulus;
 // 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));
// 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];
MyStraightLine* 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();
TriangleMeshCurviLine* 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];
MyStraightLine* 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();
TriangleMeshCurviLine* 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));


// 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;
 // 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
 Vector<TriangleMeshOpenCurve*> inner_boundary_pt;
 inner_boundary_pt.push_back(new TriangleMeshOpenCurve(internal_polyline_pt));

 // Define coordinates of a point inside the rib
 Vector<double> rib_center(2);
 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);
 // Set error estimator
 Solid_mesh_pt->spatial_error_estimator_pt()=new Z2ErrorEstimator;
#else
 // Build the mesh
 Solid_mesh_pt=new
  TriangleMesh<ELASTICITY_ELEMENT>(triangle_mesh_parameters);
#endif
 // Let's have a look where the boundaries are
 Solid_mesh_pt->output("solid_mesh.dat");
 Solid_mesh_pt->output_boundaries("solid_mesh_boundary.dat");

 // Construct the traction element mesh
 Traction_mesh_pt=new Mesh;
 create_traction_elements();

 // Solid mesh is first sub-mesh
 add_sub_mesh(Solid_mesh_pt);
 // Add traction sub-mesh
 add_sub_mesh(Traction_mesh_pt);
 // Build combined "global" mesh
 build_global_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 problem setup
 complete_problem_setup();
 //Assign equation numbers
 cout << assign_eqn_numbers() << std::endl;
 // Set output directory
 Doc_info.set_directory(Global_Parameters::Directory);
} //end_of_constructor
//====================start_of_complete_problem_setup====================
/// Complete problem setup
//=======================================================================
template<class ELASTICITY_ELEMENT>
void RingWithTRibProblem<ELASTICITY_ELEMENT>::complete_problem_setup()
{
#ifdef ADAPTIVE
 // Min element size allowed during adaptation
 if (!CommandLineArgs::command_line_flag_has_been_set("--validation"))
  {
   Solid_mesh_pt->min_element_size()=1.0e-5;
  }
#endif
 //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_region[r];
    }
  }
 // 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);
   }
 }
}
//====================start_of_actions_before_adapt=======================
/// Actions before adapt: Wipe the mesh of traction elements
//=======================================================================
template<class ELASTICITY_ELEMENT>
void RingWithTRibProblem<ELASTICITY_ELEMENT>::actions_before_adapt()
{
 // Kill the traction elements and wipe surface mesh
 delete_traction_elements();

 // 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 mesh of traction elements
//=======================================================================
template<class ELASTICITY_ELEMENT>
void RingWithTRibProblem<ELASTICITY_ELEMENT>::actions_after_adapt()
{
 // Create traction elements from all elements that are
 // adjacent to FSI boundaries and add them to surface meshes
 create_traction_elements();

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

 // Complete problem setup
 complete_problem_setup();

}// end of actions_after_adapt
//============start_of_create_traction_elements============================
/// Create traction elements
```

```
//========================================================================
template<class ELASTICITY_ELEMENT>
void RingWithTRibProblem<ELASTICITY_ELEMENT>::create_traction_elements()
{
 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
    TimeHarmonicLinearElasticityTractionElement<ELASTICITY_ELEMENT>* el_pt=
     new TimeHarmonicLinearElasticityTractionElement<ELASTICITY_ELEMENT>
     (bulk_elem_pt,face_index);

    // Add to mesh
    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_delete_traction_elements==============================
/// Delete traction elements and wipe the  traction meshes
//========================================================================
template<class ELASTICITY_ELEMENT>
void RingWithTRibProblem<ELASTICITY_ELEMENT>::delete_traction_elements()
{
 // How many surface elements are in the surface mesh
 unsigned n_element = Traction_mesh_pt->nelement();

 // Loop over the surface elements
 for(unsigned e=0;e<n_element;e++)
  {
   // Kill surface element
   delete Traction_mesh_pt->element_pt(e);
  }

 // Wipe the mesh
 Traction_mesh_pt->flush_element_and_node_storage();
} // end of delete_traction_elements
//==============start_doc=====================================
/// Doc the solution
//========================================================================
template<class ELASTICITY_ELEMENT>
void RingWithTRibProblem<ELASTICITY_ELEMENT>::doc_solution()
{
 ofstream some_file;
 char filename[100];
 // Number of plot points
 unsigned n_plot=5;
 // 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 traction elements
 //-------------------------
 sprintf(filename,"%s/traction_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Traction_mesh_pt->output(some_file,n_plot);
 some_file.close();
 // Output regions
 //---------------
 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();
  }
 // Output norm of solution (to allow validation of solution even
 // if triangle generates a slightly different mesh)
 sprintf(filename,"%s/norm%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 double norm=0.0;
 unsigned nel=Solid_mesh_pt->nelement();
 for (unsigned e=0;e<nel;e++)
  {
   double el_norm=0.0;
   Solid_mesh_pt->compute_norm(el_norm);
   norm+=el_norm;
  }
 some_file « norm « std::endl;
 // Increment label for output files
 Doc_info.number()++;
} //end doc
//=======start_of_main====================================================
/// Driver for annular disk loaded by pressure
//========================================================================
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

#ifdef ADAPTIVE
 // Max. number of adaptations
 unsigned max_adapt=3;
 CommandLineArgs::specify_command_line_flag("--max_adapt",&max_adapt);
#endif
 // Peakiness parameter for loading
 CommandLineArgs::specify_command_line_flag("--alpha",
                                            &Global_Parameters::Alpha);
 // Validation run?
 CommandLineArgs::specify_command_line_flag("--validation");
 // Parse command line
 CommandLineArgs::parse_and_assign();

 // Doc what has actually been specified on the command line
 CommandLineArgs::doc_specified_flags();
#ifdef ADAPTIVE
 //Set up the problem
 RingWithTRibProblem<ProjectableTimeHarmonicLinearElasticityElement
                     <TTimeHarmonicLinearElasticityElement<2,3> > >
  problem;
#else
 //Set up the problem
 RingWithTRibProblem<TTimeHarmonicLinearElasticityElement<2,3> > problem;
#endif
 // Initial values for parameter values
 Global_Parameters::P=0.1;
 //Parameter incrementation
 unsigned nstep=2;
 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.
   problem.newton_solve(max_adapt);
#else
   // Solve the problem using Newton's method
   problem.newton_solve();
#endif

   // Doc solution
   problem.doc_solution();

   // Now reduce the stiffness of the rib and set its inertia to
   // zero, then re-solve. Resulting displacement field should
   // be approximately axisymmetric in the annular region.
   Global_Parameters::E_pt[1]->update_constitutive_parameters(
    Global_Parameters::Nu,1.0e-10);
   Global_Parameters::Omega_sq_region[1]=0.0;
  }
```

```
} //end of main
```

1.5 Source files for this tutorial

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

  demo_drivers/time_harmonic_linear_elasticity/elastic_annulus

- The driver code is:

  demo_drivers/time_harmonic_linear_elasticity/elastic_↩
  annulus/unstructured_time_harmonic_elastic_annulus.cc

1.6 PDF file

A pdf version of this document is available.