Chapter 1

A real fluid-structure interaction problem: Finite Reynolds number flow in an oscillating elastic ring.

Our first "real" fluid-structure interaction problem: We study the finite-Reynolds number internal flow generated by the motion of an oscillating elastic ring and compare the results against asymptotic predictions.

```
//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) 2006-2026 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 2D Navier Stokes flow interacting with an elastic ring

// Oomph-lib include files
#include "generic.h"
#include "navier_stokes.h"
#include "beam.h"

//Need to include templated meshes, so that all functions
//are instantiated for our particular element types.
#include "meshes/quarter_circle_sector_mesh.h"
#include "meshes/one_d_lagrangian_mesh.h"

using namespace std;

using namespace oomph;

//============================================================================
/// Namespace for physical parameters
//============================================================================
namespace Global_Physical_Variables
{

 // Independent parameters:
 //------------------------

 /// Square of Womersly number (a frequency parameter)
```

```
 double Alpha_sq=50.0;

 /// Density ratio of the solid and the fluid
 double Density_ratio=1.0;

 /// External Pressure
 double Pext=0.0;

 /// Poisson ratio
 double Nu=0.49;

 /// Nondimensional thickness of the beam
 double H=0.05;

 /// Perturbation pressure
 double Pcos=0.0;


 // Dependent parameters:
 //----------------------

 /// Reynolds number
 double Re;

 /// Reynolds x Strouhal number
 double ReSt;

 /// Timescale ratio (non-dimensation density)
 double Lambda_sq;

 /// Stress ratio
 double Q;

 /// Set the parameters that are used in the code as a function
 /// of the Womersley number, the density ratio and H
 void set_params()
 {
  cout << "\n\n=========================================================" <<std::endl;
  cout << "\nSetting parameters. \n\n";
  cout << "Prescribed: Square of Womersley number: Alpha_sq = "
       << Alpha_sq << std::endl;
  cout << "            Density ratio:         Density_ratio = "
       << Density_ratio << std::endl;
  cout << "            Wall thickness:                   H = "
       << H << std::endl;
  cout << "            Poisson ratio:                   Nu = "
       << Nu << std::endl;
  cout << "            Pressure perturbation:          Pcos = "
       << Pcos << std::endl;


  Q=1.0/12.0*pow(H,3)/Alpha_sq;
  cout << "\nDependent:  Stress ratio:                     Q = "
       << Q << std::endl;

  Lambda_sq=1.0/12.0*pow(H,3)*Density_ratio;
  cout << "            Timescale ratio:           Lambda_sq = "
       << Lambda_sq << std::endl;

  Re=Alpha_sq;
  cout << "            Reynolds number:                  Re = "
       << Re << std::endl;

  ReSt=Re;
  cout << "          Womersley number:                ReSt = "
       << ReSt << std::endl;
  cout << "\n=========================================================\n\n"
       <<std::endl;

 }

 /// Non-FSI load function, a constant external pressure plus
 /// a (small) sinusoidal perturbation of wavenumber two.
 void pcos_load(const Vector<double>& xi, const Vector<double> &x,
                const Vector<double>& N, Vector<double>& load)
 {
  for(unsigned i=0;i<2;i++)
   {load[i] = (Pext - Pcos*cos(2.0*xi[0]))*N[i];}
 }

}


//==========================================================================
/// FSI Ring problem: a fluid-structure interaction problem in which
/// a viscous fluid bounded by an initially circular beam is set into motion
/// by a small sinusoidal perturbation of the beam (the domain boundary).
```

```
//============================================================================
class FSIRingProblem : public Problem
{
 /// There are very few element types that will work for this problem.
 /// Rather than passing the element type as a template parameter to the
 /// problem, we choose instead to use a typedef to specify the
 /// particular element fluid used.
 typedef AlgebraicElement<RefineableQCrouzeixRaviartElement<2> > FLUID_ELEMENT;

 /// Typedef to specify the solid element used
 typedef FSIHermiteBeamElement SOLID_ELEMENT;

public:

 /// Constructor: Number of elements in wall mesh, amplitude of the
 /// initial wall deformation, amplitude of pcos perturbation and its duration.
 FSIRingProblem(const unsigned &nelement_wall,
                const double& eps_ampl, const double& pcos_initial,
                const double& pcos_duration);

 /// Update after solve (empty)
 void actions_after_newton_solve() {}

 /// Update before solve (empty)
 void actions_before_newton_solve() {}

 /// Update the problem specs before checking Newton
 /// convergence
 void actions_before_newton_convergence_check()
 {
  // Update the fluid mesh -- auxiliary update function for algebraic
  // nodes automatically updates no slip condition.
  Fluid_mesh_pt->node_update();
 }

 /// Update the problem specs after adaptation:
 void actions_after_adapt()
  {
   // The functions used to update the no slip boundary conditions
   // must be set on any new nodes that have been created during the
   // mesh adaptation process.
   // There is no mechanism by which auxiliary update functions
   // are copied to newly created nodes.
   // (because, unlike boundary conditions, they don't occur exclusively
   //  at boundaries)

   // The no-slip boundary is boundary 1 of the mesh
   // Loop over the nodes on this boundary and reset the auxilliary
   // node update function
   unsigned n_node = Fluid_mesh_pt->nboundary_node(1);
   for (unsigned n=0;n<n_node;n++)
    {
     Fluid_mesh_pt->boundary_node_pt(1,n)->set_auxiliary_node_update_fct_pt(
      FSI_functions::apply_no_slip_on_moving_wall);
    }

   // (Re-)setup fsi: Work out which fluid dofs affect wall elements
   // the correspondance between wall dofs and fluid elements is handled
   // during the remeshing, but the "reverse" association must be done
   // separately.
   // We need to set up the interaction every time because the fluid element
   // adjacent to a given solid element's integration point may have changed
   // We pass the boundary between the fluid and solid meshes and pointers
   // to the meshes. The interaction boundary is boundary 1 of the 2D
   // fluid mesh.
   FSI_functions::setup_fluid_load_info_for_solid_elements<FLUID_ELEMENT,2>
    (this,1,Fluid_mesh_pt,Wall_mesh_pt);
  }

 /// Doc solution: Pass number of timestep, i (we append to tracefile
 /// after every timestep but do a full doc only at certain intervals),
 /// DocInfo object and tracefile
 void doc_solution(const unsigned& i, DocInfo& doc_info, ofstream& trace_file);

 /// Do dynamic run
 void dynamic_run();

private:

 /// Setup initial condition for both domains
 void set_initial_condition();

 /// Setup initial condition for wall
 void set_wall_initial_condition();

 /// Setup initial condition for fluid
 void set_fluid_initial_condition();
```

```
 /// Element used for documenting displacement
 SOLID_ELEMENT* Doc_displacement_elem_pt;

 /// Pointer to wall mesh
 OneDLagrangianMesh<SOLID_ELEMENT> *Wall_mesh_pt;

 /// Pointer to fluid mesh
 AlgebraicRefineableQuarterCircleSectorMesh<FLUID_ELEMENT> *Fluid_mesh_pt;

 /// Pointer to geometric object that represents the undeformed wall shape
 GeomObject* Undef_geom_pt;

 /// Pointer to wall timestepper
 Newmark<2>* Wall_time_stepper_pt;

 /// Pointer to fluid timestepper
 BDF<2>* Fluid_time_stepper_pt;

 /// Pointer to node on coarsest mesh on which velocity is traced
 Node* Veloc_trace_node_pt;

 /// Amplitude of initial deformation
 double Eps_ampl;

 /// Initial pcos
 double Pcos_initial;

 /// Duration of initial pcos
 double Pcos_duration;

};


//============================================================
/// Setup initial condition: When we're done here, all variables
/// represent the state at the initial time.
//============================================================
void FSIRingProblem::set_initial_condition()
{

 cout << "Setting wall ic" << std::endl;
 set_wall_initial_condition();

 cout << "Setting fluid ic" << std::endl;
 set_fluid_initial_condition();

}


//============================================================
/// Setup initial condition for fluid: Impulsive start
//============================================================
void FSIRingProblem::set_fluid_initial_condition()
{

 // Update fluid domain: Careful!!! This also applies the no slip conditions
 // on all nodes on the wall! Since the wall might have moved since
 // we created the mesh; we're therefore imposing a nonzero
 // velocity on these nodes. Must wipe this afterwards (done
 // by setting *all* velocities to zero) otherwise we get
 // an impulsive start from a very bizarre initial velocity
 // field! [Yes, it took me a while to figure this out...]
 Fluid_mesh_pt->node_update();

 // Assign initial values for the velocities;
 // pressures don't carry a time history and can be left alone.

 //Find number of nodes in fluid mesh
 unsigned n_node = Fluid_mesh_pt->nnode();

 // Loop over the nodes to set initial guess everywhere
 for(unsigned n=0;n<n_node;n++)
  {
   // Loop over velocity directions: Impulsive initial start from
   // zero velocity!
   for(unsigned i=0;i<2;i++)
    {
     Fluid_mesh_pt->node_pt(n)->set_value(i,0.0);
    }
  }

 // Do an impulsive start with the assigned velocity field
```

```
 Fluid_mesh_pt->assign_initial_values_impulsive();

}


//========================================================================
/// Setup initial condition: Impulsive start either from
/// deformed or undeformed wall shape.
//========================================================================
void FSIRingProblem::set_wall_initial_condition()
{

 // Geometric object that specifies the initial conditions:
 // A ring that is bucked in a 2-lobed mode
 GeomObject* ic_geom_object_pt=
  new PseudoBucklingRing(Eps_ampl,Global_Physical_Variables::H,2,2,
                         Wall_time_stepper_pt);
 // Assign period of oscillation of the geometric object
 static_cast<PseudoBucklingRing*>(ic_geom_object_pt)->set_T(1.0);
 //Set initial time (to deform wall into max. amplitude)
 double time=0.25;
 // Assign initial radius of the object
 static_cast<PseudoBucklingRing*>(ic_geom_object_pt)->set_R_0(1.00);
 // Setup object that specifies the initial conditions:
 SolidInitialCondition* IC_pt = new SolidInitialCondition(ic_geom_object_pt);
 // Assign values of positional data of all elements on wall mesh
 // so that the wall deforms into the shape specified by IC object.
 SolidMesh::Solid_IC_problem.set_static_initial_condition(
  this,Wall_mesh_pt,IC_pt,time);
}


//========================================================================
/// Document solution: Pass number of timestep, i; we append to trace file
/// at every timestep and do a full doc only after a certain number
/// of steps.
//========================================================================
void FSIRingProblem::doc_solution(const unsigned& i,
  DocInfo& doc_info, ofstream& trace_file)
{

  // Full doc every nskip steps
  unsigned nskip=1; // ADJUST

  // If we at an integer multiple of nskip, full documentation.
  if (i%nskip==0)
   {
    doc_info.enable_doc();
    cout << "Full doc step " <<  doc_info.number()
         << " for time " << time_stepper_pt()->time_pt()->time() << std::endl;
   }
  //Otherwise, just output the trace file
  else
   {
    doc_info.disable_doc();
    cout << "Only trace for time "
         << time_stepper_pt()->time_pt()->time() << std::endl;
   }


  // If we are at a full documentation step, output the fluid solution
  if (doc_info.is_doc_enabled())
   {
    //Variables used in the output file.
    ofstream some_file; char filename[100];
    //Construct the output filename from the doc_info number and the
    //output directory
    snprintf(filename, sizeof(filename), "%s/soln%i.dat",doc_info.directory().c_str(),
            doc_info.number());
    //Open the output file
    some_file.open(filename);
    /// Output the solution using 5x5 plot points
    Fluid_mesh_pt->output(some_file,5);
    //Close the output file
    some_file.close();
   }
  //Temporary vector to give the local coordinate at which to document
  //the wall displacment
  Vector<double> s(1,1.0);
  // Write to the trace file:
  trace_file << time_pt()->time()
   //Document the displacement at the end of the the chosen element
             << " " << Doc_displacement_elem_pt->interpolated_x(s,1)
             << " " << Veloc_trace_node_pt->x(0)
             << " " << Veloc_trace_node_pt->x(1)
             << " " << Veloc_trace_node_pt->value(0)
             << " " << Veloc_trace_node_pt->value(1)
```

```
             « " " « Fluid_mesh_pt->nelement()
             « " " « ndof()
             « " " « Fluid_mesh_pt->nrefinement_overruled()
             « " " « Fluid_mesh_pt->max_error()
             « " " « Fluid_mesh_pt->min_error()
             « " " « Fluid_mesh_pt->max_permitted_error()
             « " " « Fluid_mesh_pt->min_permitted_error()
             « " " « Fluid_mesh_pt->max_keep_unrefined();

  // Output the number of the corresponding full documentation
  // file number (or -1 if no full doc was made)
  if (doc_info.is_doc_enabled())
   {trace_file « " " «doc_info.number()  « " ";}
  else {trace_file « " " «-1  « " ";}

  //End the trace file
  trace_file « std::endl;

  // Increment counter for full doc
  if (doc_info.is_doc_enabled()) {doc_info.number()++;}
}

//========================================================================
/// Constructor for FSI ring problem. Pass number of wall elements
/// and length of wall (in Lagrangian coordinates)  amplitude of
/// initial deformation, pcos perturbation and duration.
//========================================================================
FSIRingProblem::FSIRingProblem(const unsigned& N,
               const double& eps_ampl, const double& pcos_initial,
               const double& pcos_duration) :
 Eps_ampl(eps_ampl), Pcos_initial(pcos_initial),
 Pcos_duration(pcos_duration)
{
 //--------------------------------------------------------
 // Create timesteppers
 //--------------------------------------------------------
 // Allocate the wall timestepper and add it to the problem's vector
 // of timesteppers
 Wall_time_stepper_pt = new Newmark<2>;
 add_time_stepper_pt(Wall_time_stepper_pt);

 // Allocate the fluid timestepper and add it to the problem's Vector
 // of timesteppers
 Fluid_time_stepper_pt = new BDF<2>;
 add_time_stepper_pt(Fluid_time_stepper_pt);

 //--------------------------------------------------------
 // Create the wall mesh
 //--------------------------------------------------------

 // Undeformed wall is an elliptical ring
 Undef_geom_pt = new Ellipse(1.0,1.0);

 //Length of wall in Lagrangian coordinates
 double L = 2.0*atan(1.0);

 //Now create the (Lagrangian!) mesh
 Wall_mesh_pt = new
  OneDLagrangianMesh<SOLID_ELEMENT>(N,L,Undef_geom_pt,Wall_time_stepper_pt);

 //--------------------------------------------------------
 // Set the boundary conditions for wall mesh (problem)
 //--------------------------------------------------------
 // Bottom boundary: (Boundary 0)
 // No vertical displacement
 Wall_mesh_pt->boundary_node_pt(0,0)->pin_position(1);
 // Zero slope: Pin type 1 dof for displacement direction 0
 Wall_mesh_pt->boundary_node_pt(0,0)->pin_position(1,0);
 // Top boundary: (Boundary 1)
 // No horizontal displacement
 Wall_mesh_pt->boundary_node_pt(1,0)->pin_position(0);
 // Zero slope: Pin type 1 dof for displacement direction 1
 Wall_mesh_pt->boundary_node_pt(1,0)->pin_position(1,1);


 //--------------------------------------------------------
 // Create the fluid mesh:
 //--------------------------------------------------------

 // Fluid mesh is suspended from wall between the following Lagrangian
 // coordinates:
 double xi_lo=0.0;
 double xi_hi=L;

 // Fractional position of dividing line for two outer blocks in mesh
 double fract_mid=0.5;
 //Create a geometric object that represents the wall geometry from the
```

```
//wall mesh (one Lagrangian, two Eulerian coordinates).
MeshAsGeomObject *wall_mesh_as_geometric_object_pt
 = new MeshAsGeomObject(Wall_mesh_pt);

// Build fluid mesh using the wall mesh as a geometric object
Fluid_mesh_pt = new AlgebraicRefineableQuarterCircleSectorMesh<FLUID_ELEMENT >
 (wall_mesh_as_geometric_object_pt,
  xi_lo,fract_mid,xi_hi,Fluid_time_stepper_pt);

// Set the error estimator
Z2ErrorEstimator* error_estimator_pt=new Z2ErrorEstimator;
Fluid_mesh_pt->spatial_error_estimator_pt()=error_estimator_pt;

// Extract pointer to node at center of mesh
unsigned nnode=Fluid_mesh_pt->finite_element_pt(0)->nnode();
Veloc_trace_node_pt=Fluid_mesh_pt->finite_element_pt(0)->node_pt(nnode-1);
//-----------------------------------------------------
// Set the fluid boundary conditions
//-----------------------------------------------------

// Bottom boundary (boundary 0):
{
 unsigned n_node = Fluid_mesh_pt->nboundary_node(0);
 for (unsigned n=0;n<n_node;n++)
  {
   // Pin vertical velocity
   Fluid_mesh_pt->boundary_node_pt(0,n)->pin(1);
  }
}

// Ring boundary (boundary 1):
// No slip; this also implies that the velocity needs
// to be updated in response to wall motion
{
 unsigned n_node = Fluid_mesh_pt->nboundary_node(1);
 for (unsigned n=0;n<n_node;n++)
  {
   // Which node are we dealing with?
   Node* node_pt=Fluid_mesh_pt->boundary_node_pt(1,n);

    // Set auxiliary update function pointer
   node_pt->set_auxiliary_node_update_fct_pt(
    FSI_functions::apply_no_slip_on_moving_wall);

   // Pin both velocities
   for(unsigned i=0;i<2;i++) {node_pt->pin(i);}
  }
}

// Left boundary (boundary 2):
 {
  unsigned n_node = Fluid_mesh_pt->nboundary_node(2);
  for (unsigned n=0;n<n_node;n++)
   {
    // Pin horizontal velocity
      Fluid_mesh_pt->boundary_node_pt(2,n)->pin(0);
   }
 }


//-----------------------------------------------------
// Add the submeshes and build global mesh
// ----------------------------------------------------

// Wall mesh
add_sub_mesh(Wall_mesh_pt);

//Fluid mesh
add_sub_mesh(Fluid_mesh_pt);

// Combine all submeshes into a single Mesh
build_global_mesh();

//-----------------------------------------------------------
// Finish problem setup
// ----------------------------------------------------------

//Find number of elements in fluid mesh
unsigned n_element = Fluid_mesh_pt->nelement();

// Loop over the fluid elements to set up element-specific
// things that cannot be handled by constructor
for(unsigned e=0;e<n_element;e++)
 {
  // Upcast from FiniteElement to the present element
  FLUID_ELEMENT *el_pt
```

```
    = dynamic_cast<FLUID_ELEMENT*>(Fluid_mesh_pt->element_pt(e));

   //Set the Reynolds number, etc
   el_pt->re_pt() = &Global_Physical_Variables::Re;
   el_pt->re_st_pt() = &Global_Physical_Variables::ReSt;


   el_pt->evaluate_shape_derivs_by_direct_fd();

//   el_pt->evaluate_shape_derivs_by_chain_rule();
//   el_pt->enable_always_evaluate_dresidual_dnodal_coordinates_by_fd();

//    if (e==0)
//     {
//      el_pt->disable_always_evaluate_dresidual_dnodal_coordinates_by_fd();
//     }
//    else
//     {
//      el_pt->enable_always_evaluate_dresidual_dnodal_coordinates_by_fd();
//     }

   //el_pt->evaluate_shape_derivs_by_direct_fd();

  }
 //Loop over the solid elements to set physical parameters etc.
 unsigned n_wall_element = Wall_mesh_pt->nelement();
 for(unsigned e=0;e<n_wall_element;e++)
  {
   //Cast to proper element type
   SOLID_ELEMENT *el_pt = dynamic_cast<SOLID_ELEMENT*>(
    Wall_mesh_pt->element_pt(e));

   //Set physical parameters for each element:
   el_pt->h_pt() = &Global_Physical_Variables::H;
   el_pt->lambda_sq_pt() = &Global_Physical_Variables::Lambda_sq;

   //Function that specifies the external load Vector
   el_pt->load_vector_fct_pt() = &Global_Physical_Variables::pcos_load;

   // Function that specifies the load ratios
   el_pt->q_pt() = &Global_Physical_Variables::Q;

   //Assign the undeformed beam shape
   el_pt->undeformed_beam_pt() = Undef_geom_pt;
  }
 // Establish control displacment: (even if no displacement control is applied
 // we still want to doc the displacement at the same point)

 // Choose element: (This is the last one)
 Doc_displacement_elem_pt = dynamic_cast<SOLID_ELEMENT*>(
  Wall_mesh_pt->element_pt(n_wall_element-1));

 // Setup fsi: Work out which fluid dofs affect the wall elements
 // the correspondance between wall dofs and fluid elements is handled
 // during the remeshing, but the "reverse" association must be done
 // separately.
 // We pass the boundary between the fluid and solid meshes and pointers
 // to the meshes. The interaction boundary is boundary 1 of the
 // 2D fluid mesh.
 FSI_functions::setup_fluid_load_info_for_solid_elements<FLUID_ELEMENT,2>
  (this,1,Fluid_mesh_pt,Wall_mesh_pt);

 // Do equation numbering
 cout << "# of dofs " << assign_eqn_numbers() << std::endl;
}


//==========================================================================
/// Solver loop to perform unsteady run
//==========================================================================
void FSIRingProblem::dynamic_run()
{
 // Setup documentation
 //-----------------------------------------------------------------

 /// Label for output
 DocInfo doc_info;

 // Output directory
 doc_info.set_directory("RESLT");

 // Step number
 doc_info.number()=0;

 //Open a trace file
 ofstream trace_file("RESLT/trace_ring.dat");
 // Write header for trace file
```

```
trace_file « "VARIABLES=\"time\",\"V_c_t_r_l\"";
trace_file « ",\"x<sub>1</sub><sup>(track)</sup>\"";
trace_file « ",\"x<sub>2</sub><sup>(track)</sup>\"";
trace_file « ",\"u<sub>1</sub><sup>(track)</sup>\"";
trace_file « ",\"u<sub>2</sub><sup>(track)</sup>\"";
trace_file « ",\"N<sub>element</sub>\"";
trace_file « ",\"N<sub>dof</sub>\"";
trace_file « ",\"# of under-refined elements\"";
trace_file « ",\"max. error\"";
trace_file « ",\"min. error\"";
trace_file « ",\"max. permitted error\"";
trace_file « ",\"min. permitted error\"";
trace_file « ",\"max. permitted # of unrefined elements\"";
trace_file « ",\"doc number\"";
trace_file « std::endl;

// Initialise timestepping
// ------------------------------------------------------------

// Number of steps
unsigned nstep=300;

// Nontrivial command line input: validation: only do three steps
if (CommandLineArgs::Argc>1)
 {
  nstep=1;
  cout « "Only doing nstep steps for validation: " « nstep « std::endl;
 }

// Set initial timestep
double dt=0.004;

// Set initial value for dt  -- also assigns weights to the timesteppers
initialise_dt(dt);

// Set physical parameters
//------------------------------------------------------------
using namespace Global_Physical_Variables;

// Set Womersley number
Alpha_sq=100.0; // 50.0; // ADJUST

// Set density ratio
Density_ratio=10.0; // 0.0; ADJUST

// Wall thickness
H=1.0/20.0;

// Set external pressure
Pext=0.0;

/// Perturbation pressure
Pcos=Pcos_initial;

// Assign/doc corresponding computational parameters
set_params();


// Refine uniformly and assign initial conditions
//------------------------------------------------------------------

// Refine the problem uniformly
refine_uniformly();
refine_uniformly();

// This sets up the solution at the initial time
set_initial_condition();

// Set targets for spatial adptivity
//------------------------------------------------------------------

// Max. and min. error for adaptive refinement/unrefinement
Fluid_mesh_pt->max_permitted_error()=1.0e-2;
Fluid_mesh_pt->min_permitted_error()=1.0e-3;

// Don't allow refinement to drop under given level
Fluid_mesh_pt->min_refinement_level()=2;

// Don't allow refinement beyond given level
Fluid_mesh_pt->max_refinement_level()=6;

// Don't bother adapting the mesh if no refinement is required
// and if less than ... elements are to be merged.
Fluid_mesh_pt->max_keep_unrefined()=20;

// Doc refinement targets
```

```
 Fluid_mesh_pt->doc_adaptivity_targets(cout);


 // Do the timestepping
 //----------------------------------------------------------------

 // Reset initial conditions after refinement for first step only
 bool first=true;

 //Output initial data
 doc_solution(0,doc_info,trace_file);


//  {
//   unsigned nel=Fluid_mesh_pt->nelement();
//   for (unsigned e=0;e<nel;e++)
//    {
//     std::cout << "\n\nEl: " << e << std::endl << std::endl;
//     FiniteElement* el_pt=Fluid_mesh_pt->finite_element_pt(e);
//     unsigned n_dof=el_pt->ndof();
//     Vector<double> residuals(n_dof);
//     DenseDoubleMatrix jacobian(n_dof,n_dof);
//     el_pt->get_jacobian(residuals,jacobian);
//    }
//   exit(0);
//  }

 // Time integration loop
 for(unsigned i=1;i<=nstep;i++)
  {
   // Switch doc off during solve
   doc_info.disable_doc();

   // Solve
   unsigned max_adapt=1;
   unsteady_newton_solve(dt,max_adapt,first);

   // Now we've done the first step
   first=false;

   // Doc solution
   doc_solution(i,doc_info,trace_file);

   /// Switch off perturbation pressure
   if (time_pt()->time()>Pcos_duration) {Pcos=0.0;}

  }

}


//======================================================================
/// Driver for fsi ring test problem
//======================================================================
int main(int argc, char* argv[])
{

 // Store command line arguments
 CommandLineArgs::setup(argc,argv);
 // Number of elements
 unsigned nelem = 13;

 /// Perturbation pressure
 double pcos_initial=1.0e-6; // ADJUST

 /// Duration of initial pcos perturbation
 double pcos_duration=0.3; // ADJUST

 /// Amplitude of initial deformation
 double eps_ampl=0.0; // ADJUST

 //Set up the problem
 FSIRingProblem problem(nelem,eps_ampl,pcos_initial,pcos_duration);

 // Do parameter study
 problem.dynamic_run();

}
```

1.1 PDF file

A pdf version of this document is available. \