Chapter 1

Demo problem: Small-amplitude non-axisymmetric oscillations of a thin-walled elastic ring

Detailed documentation to be written. Here's the already fairly well documented driver code...

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

//OOMPH-LIB includes
#include "generic.h"
#include "beam.h"
#include "meshes/one_d_lagrangian_mesh.h"

using namespace std;

using namespace oomph;

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


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

 /// Flag for long/short run: Default =  perform long run
 unsigned Long_run_flag=1;
```

```
 /// \short Flag for fixed timestep: Default = fixed timestep
 unsigned Fixed_timestep_flag=1;

 /// \short Boolean flag to decide if to set IC for Newmark
 /// directly or consistently : No Default
 bool Consistent_newmark_ic;

} // end of namespace

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



//==start_of_problem_class============================================
/// Oscillating ring problem: Compare small-amplitude oscillations
/// against analytical solution of the linearised equations.
//====================================================================
template<class ELEMENT, class TIMESTEPPER>
class ElasticRingProblem : public Problem
{

public:

 /// \short Constructor: Number of elements, length of domain, flag for
 /// setting Newmark IC directly or consistently
 ElasticRingProblem(const unsigned &N, const double &L);

 /// Access function for the mesh
 OneDLagrangianMesh<ELEMENT>* mesh_pt()
  {
   return dynamic_cast<OneDLagrangianMesh<ELEMENT>*>(Problem::mesh_pt());
  }

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

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

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

 /// Do unsteady run
 void unsteady_run();

private:

 /// Length of domain (in terms of the Lagrangian coordinates)
 double Length;

 /// \short In which element are we applying displacement control?
 /// (here only used for doc of radius)
 ELEMENT* Displ_control_elem_pt;

 /// At what local coordinate are we applying displacement control?
 Vector<double> S_displ_control;

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

 /// \short Pointer to object that specifies the initial condition
 SolidInitialCondition* IC_pt;

 /// Trace file for recording control data
 ofstream Trace_file;
}; // end of problem class



//===start_of_constructor=============================================
/// Constructor for elastic ring problem
//====================================================================
template<class ELEMENT,class TIMESTEPPER>
ElasticRingProblem<ELEMENT,TIMESTEPPER>::ElasticRingProblem
(const unsigned& N, const double& L)
 : Length(L)
{

 //Allocate the timestepper -- This constructs the time object as well
 add_time_stepper_pt(new TIMESTEPPER());

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

```
 //Now create the (Lagrangian!) mesh
 Problem::mesh_pt() = new OneDLagrangianMesh<ELEMENT>(
  N,L,Undef_geom_pt,Problem::time_stepper_pt());

 // Boundary condition:

 // Bottom:
 unsigned ibound=0;
 // No vertical displacement
 mesh_pt()->boundary_node_pt(ibound,0)->pin_position(1);
 // Zero slope: Pin type 1 dof for displacement direction 0
 mesh_pt()->boundary_node_pt(ibound,0)->pin_position(1,0);

 // Top:
 ibound=1;
 // No horizontal displacement
 mesh_pt()->boundary_node_pt(ibound,0)->pin_position(0);
 // Zero slope: Pin type 1 dof for displacement direction 1
 mesh_pt()->boundary_node_pt(ibound,0)->pin_position(1,1);


 // Resize vector of local coordinates for control displacement
 // (here only used to identify the point whose displacement we're
 // tracing)
 S_displ_control.resize(1);

 // Complete build of all elements so they are fully functional
 // -----------------------------------------------------------

 // Find number of elements in mesh
 unsigned Nelement = mesh_pt()->nelement();

 // Loop over the elements to set pointer to undeformed wall shape
 for(unsigned i=0;i<Nelement;i++)
  {
   // Cast to proper element type
   ELEMENT *elem_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));

   // Assign the undeformed surface
   elem_pt->undeformed_beam_pt() = Undef_geom_pt;
  }

 // Establish control displacment: (even though no displacement
 // control is applied we still want to doc the displacement at the same point)

 // Choose element: (This is the last one)
 Displ_control_elem_pt=dynamic_cast<ELEMENT*>(
  mesh_pt()->element_pt(Nelement-1));

 // Fix/doc the displacement in the vertical (1) direction at right end of
 // the control element
 S_displ_control[0]=1.0;

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

 // Geometric object that specifies the initial conditions
 double eps_buckl=1.0e-2;
 double HoR=dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(0))->h();
 unsigned n_buckl=2;
 unsigned imode=2;
 GeomObject* ic_geom_object_pt=
  new PseudoBucklingRing(eps_buckl,HoR,n_buckl,imode,
                         Problem::time_stepper_pt());

 // Setup object that specifies the initial conditions:
 IC_pt = new SolidInitialCondition(ic_geom_object_pt);

} // end of constructor


//===start_of_doc_solution================================================
/// Document solution
//========================================================================
template<class ELEMENT, class TIMESTEPPER>
void ElasticRingProblem<ELEMENT, TIMESTEPPER>::doc_solution
      (
 DocInfo& doc_info)
{

 cout << "Doc-ing step " <<  doc_info.number()
      << " for time " << time_stepper_pt()->time_pt()->time() << std::endl;


 // Loop over all elements to get global kinetic and potential energy
 unsigned Nelem=mesh_pt()->nelement();
```

```
double global_kin=0;
double global_pot=0;
double pot,kin;
for (unsigned ielem=0;ielem<Nelem;ielem++)
 {
  dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(ielem))->get_energy(pot,kin);
  global_kin+=kin;
  global_pot+=pot;
 }


// Control displacement for initial condition object
Vector<double> xi_ctrl(1);
Vector<double> posn_ctrl(2);

// Lagrangian coordinate of control point
xi_ctrl[0]=Displ_control_elem_pt->interpolated_xi(S_displ_control,0);

// Get position
IC_pt->geom_object_pt()->position(xi_ctrl,posn_ctrl);

// Write trace file: Time, control position, energies
Trace_file << time_pt()->time()  << " "
           << Displ_control_elem_pt->interpolated_x(S_displ_control,1)
           << " " << global_pot  << " " << global_kin
           << " " << global_pot + global_kin
           << " " << posn_ctrl[1]
           << std::endl;


ofstream some_file;
char filename[100];

// Number of plot points
unsigned npts=5;

// Output solution
sprintf(filename,"%s/ring%i.dat",doc_info.directory().c_str(),
        doc_info.number());
some_file.open(filename);
mesh_pt()->output(some_file,npts);
some_file.close();

// Loop over all elements do dump out previous solutions
unsigned nsteps=time_stepper_pt()->nprev_values();
for (unsigned t=0;t<=nsteps;t++)
 {
  sprintf(filename,"%s/ring%i-%i.dat",doc_info.directory().c_str(),
          doc_info.number(),t);
  some_file.open(filename);
  unsigned Nelem=mesh_pt()->nelement();
  for (unsigned ielem=0;ielem<Nelem;ielem++)
   {
    dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(ielem))->
     output(t,some_file,npts);
   }
  some_file.close();
 }

// Output for initial condition object
sprintf(filename,"%s/ic_ring%i.dat",doc_info.directory().c_str(),
        doc_info.number());
some_file.open(filename);

unsigned nplot=1+(npts-1)*mesh_pt()->nelement();
Vector<double> xi(1);
Vector<double> posn(2);
Vector<double> veloc(2);
Vector<double> accel(2);

for (unsigned iplot=0;iplot<nplot;iplot++)
 {
  xi[0]=Length/double(nplot-1)*double(iplot);

  IC_pt->geom_object_pt()->position(xi,posn);
  IC_pt->geom_object_pt()->dposition_dt(xi,1,veloc);
  IC_pt->geom_object_pt()->dposition_dt(xi,2,accel);

  some_file << posn[0] << " " << posn[1] << " "
            << xi[0] << " "
            << veloc[0] << " " << veloc[1] << " "
            << accel[0] << " " << accel[1] << " "
            << sqrt(pow(posn[0],2)+pow(posn[1],2)) << " "
            << sqrt(pow(veloc[0],2)+pow(veloc[1],2)) << " "
            << sqrt(pow(accel[0],2)+pow(accel[1],2)) << " "
            << std::endl;
 }
```

```
 some_file.close();
} // end of doc solution



//===start_of_unsteady_run===============================================
/// Solver loop to perform unsteady run
//======================================================================
template<class ELEMENT,class TIMESTEPPER>
void ElasticRingProblem<ELEMENT,TIMESTEPPER>::unsteady_run
      ()
{

 /// Label for output
 DocInfo doc_info;

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

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

 // Set up trace file
 char filename[100];
 sprintf(filename,"%s/trace_ring.dat",doc_info.directory().c_str());
 Trace_file.open(filename);

 Trace_file <<  "VARIABLES=\"time\",\"R<sub>ctrl</sub>\",\"E<sub>pot</sub>\"";
 Trace_file << ",\"E<sub>kin</sub>\",\"E<sub>kin</sub>+E<sub>pot</sub>\"";
 Trace_file << ",\"<sub>exact</sub>R<sub>ctrl</sub>\""
            << std::endl;

 // Number of steps
 unsigned nstep=600;
 if (Global_Physical_Variables::Long_run_flag==0) {nstep=10;}

 // Initial timestep
 double dt=1.0;

 // Ratio for timestep reduction
 double timestep_ratio=1.0;
 if (Global_Physical_Variables::Fixed_timestep_flag==0) {
      timestep_ratio=0.995;}

 // Number of previous timesteps stored
 unsigned ndt=time_stepper_pt()->time_pt()->ndt();

 // Setup vector of "previous" timesteps
 Vector<double> dt_prev(ndt);
 dt_prev[0]=dt;
 for (unsigned i=1;i<ndt;i++)
  {
   dt_prev[i]=dt_prev[i-1]/timestep_ratio;
  }

 // Initialise the history of previous timesteps
 time_pt()->initialise_dt(dt_prev);

 // Initialise time
 double time0=10.0;
 time_pt()->time()=time0;

 // Setup analytical initial condition?
 if (Global_Physical_Variables::Consistent_newmark_ic)
  {
   // Note: Time has been scaled on intrinsic timescale so
   // we don't need to specify a multiplier for the inertia
   // terms (the default assignment of 1.0 is OK)
   SolidMesh::Solid_IC_problem.
    set_newmark_initial_condition_consistently(
     this,mesh_pt(),static_cast<TIMESTEPPER*>(time_stepper_pt()),IC_pt,dt);
  }
 else
  {
   SolidMesh::Solid_IC_problem.
    set_newmark_initial_condition_directly(
     this,mesh_pt(),static_cast<TIMESTEPPER*>(time_stepper_pt()),IC_pt,dt);
  }

 //Output initial data
 doc_solution(doc_info);

 // Time integration loop
 for(unsigned i=1;i<=nstep;i++)
  {
   // Solve
```

```
   unsteady_newton_solve(dt);

   // Doc solution
   doc_info.number()++;
   doc_solution(doc_info);

   // Reduce timestep
   if (time_pt()->time()<100.0) {dt=timestep_ratio*dt;}
  }

} // end of unsteady run



//===start_of_main=====================================================
/// Driver for ring that performs small-amplitude oscillations
//=====================================================================
int main(int argc, char* argv[])
{

 // Store command line arguments
 CommandLineArgs::setup(argc,argv);

 /// Convert command line arguments (if any) into flags:
 if (argc==2)
  {
   // Nontrivial command line input: Setup Newmark IC directly
   // (rather than consistently with PVD)
   if (atoi(argv[1])==1)
    {
     Global_Physical_Variables::Consistent_newmark_ic=true;
     cout << "Setting Newmark IC consistently" << std::endl;
    }
   else
    {
     Global_Physical_Variables::Consistent_newmark_ic=false
      ;
     cout << "Setting Newmark IC directly" << std::endl;
    }

   cout << "Not enough command line arguments specified -- using defaults."
        << std::endl;
  } // end of 1 argument
 else if (argc==4)
  {
   cout << "Three command line arguments specified:" << std::endl;
   // Nontrivial command line input: Setup Newmark IC directly
   // (rather than consistently with PVD)
   if (atoi(argv[1])==1)
    {
     Global_Physical_Variables::Consistent_newmark_ic=true;
     cout << "Setting Newmark IC consistently" << std::endl;
    }
   else
    {
     Global_Physical_Variables::Consistent_newmark_ic=false
      ;
     cout << "Setting Newmark IC directly" << std::endl;
    }
   // Flag for long run
   Global_Physical_Variables::Long_run_flag=atoi(argv[2]);
   // Flag for fixed timestep
   Global_Physical_Variables::Fixed_timestep_flag=atoi(argv[3
      ]);
  } // end of 3 arguments
 else
  {
   std::string error_message =
    "Wrong number of command line arguments. Specify one or three.\n";
   error_message += "Arg1: Long_run_flag [0/1]\n";
   error_message += "Arg2: Impulsive_start_flag [0/1]\n";
   error_message += "Arg3: Restart_flag [restart_file] (optional)\n";

   throw OomphLibError(error_message,
                       OOMPH_CURRENT_FUNCTION,
                       OOMPH_EXCEPTION_LOCATION);
  } // too many arguments
 cout << "Setting Newmark IC consistently: "
      <<  Global_Physical_Variables::Consistent_newmark_ic
      << std::endl;
 cout << "Long run flag: "
      <<  Global_Physical_Variables::Long_run_flag << std::endl;
 cout << "Fixed timestep flag: "
      <<  Global_Physical_Variables::Fixed_timestep_flag <<
      std::endl;

 //Length of domain
```

```
double L = MathematicalConstants::Pi/2.0;

// Number of elements
unsigned nelem = 13;

//Set up the problem
ElasticRingProblem<HermiteBeamElement,Newmark<3> >
 problem(nelem,L);

// Do unsteady run
problem.unsteady_run();

} // end of main
```

1.1 PDF file

A pdf version of this document is available.