# Chapter 1

# Demo problem: Large-amplitude shock-wave propagation in a circular disk

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// 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 large-displacement elasto-dynamic test problem:
// Circular disk impulsively loaded by compressive load.

#include <iostream>
#include <fstream>
#include <cmath>

//My own includes
#include "generic.h"
#include "solid.h"

//Need to instantiate templated mesh
#include "meshes/quarter_circle_sector_mesh.h"

using namespace std;

using namespace oomph;

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


//================================================================
/// Global variables
//================================================================
namespace Global_Physical_Variables
{
 /// Pointer to constitutive law
 ConstitutiveLaw* Constitutive_law_pt;
```

```
 /// Elastic modulus
 double E=1.0;

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

 /// Uniform pressure
 double P = 0.00;

 /// Constant pressure load
 void constant_pressure(const Vector<double> &xi,const Vector<double> &x,
                        const Vector<double> &n, Vector<double> &traction)
 {
  unsigned dim = traction.size();
  for(unsigned i=0;i<dim;i++)
   {
    traction[i] = -P*n[i];
   }
 }

}



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



//=================================================================
/// Elastic quarter circle sector mesh with functionality to
/// attach traction elements to the curved surface. We "upgrade"
/// the RefineableQuarterCircleSectorMesh to become an
/// SolidMesh and equate the Eulerian and Lagrangian coordinates,
/// thus making the domain represented by the mesh the stress-free
/// configuration.
/// \n\n
/// The member function \c make_traction_element_mesh() creates
/// a separate mesh of SolidTractionElements that are attached to the
/// mesh's curved boundary (boundary 1).
//=================================================================
template <class ELEMENT>
class ElasticRefineableQuarterCircleSectorMesh :
 public virtual RefineableQuarterCircleSectorMesh<ELEMENT>,
 public virtual SolidMesh
{


public:

 /// Constructor: Build mesh and copy Eulerian coords to Lagrangian
 /// ones so that the initial configuration is the stress-free one.
 ElasticRefineableQuarterCircleSectorMesh<ELEMENT>(GeomObject* wall_pt,
                                        const double& xi_lo,
                                        const double& fract_mid,
                                        const double& xi_hi,
                                        TimeStepper* time_stepper_pt=
                                        &Mesh::Default_TimeStepper) :
  RefineableQuarterCircleSectorMesh<ELEMENT>(wall_pt,xi_lo,fract_mid,xi_hi,
                                            time_stepper_pt)
  {
#ifdef PARANOID
   /// Check that the element type is derived from the SolidFiniteElement
   SolidFiniteElement* el_pt=dynamic_cast<SolidFiniteElement*>
    (finite_element_pt(0));
   if (el_pt==0)
    {
     throw OomphLibError(
      "Element needs to be derived from SolidFiniteElement\n",
      OOMPH_CURRENT_FUNCTION,
      OOMPH_EXCEPTION_LOCATION);
    }
#endif

   // Make the current configuration the undeformed one by
   // setting the nodal Lagrangian coordinates to their current
   // Eulerian ones
   set_lagrangian_nodal_coordinates();
  }


 /// Function to create mesh made of traction elements
 void make_traction_element_mesh(SolidMesh*& traction_mesh_pt)
  {

   // Make new mesh
```

```
  traction_mesh_pt=new SolidMesh;

  // Loop over all elements on boundary 1:
  unsigned b=1;
  unsigned n_element = this->nboundary_element(b);
  for (unsigned e=0;e<n_element;e++)
   {
    // The element itself:
    FiniteElement* fe_pt = this->boundary_element_pt(b,e);

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

    // Create new element
    traction_mesh_pt->add_element_pt(new SolidTractionElement<ELEMENT>
                                     (fe_pt,face_index));
   }
 }


 /// Function to wipe and re-create mesh made of traction elements
 void remake_traction_element_mesh(SolidMesh*& traction_mesh_pt)
  {

  // Wipe existing mesh (but don't call it's destructor as this
  // would wipe all the nodes too!)
  traction_mesh_pt->flush_element_and_node_storage();

  // Loop over all elements on boundary 1:
  unsigned b=1;
  unsigned n_element = this->nboundary_element(b);
  for (unsigned e=0;e<n_element;e++)
   {
    // The element itself:
    FiniteElement* fe_pt = this->boundary_element_pt(b,e);

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

    // Create new element
    traction_mesh_pt->add_element_pt(new SolidTractionElement<ELEMENT>
                                     (fe_pt,face_index));
   }
  }


};




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



//=========================================================================
/// "Shock" wave propagates through an impulsively loaded
/// circular disk.
//=========================================================================
template<class ELEMENT, class TIMESTEPPER>
class DiskShockWaveProblem : public Problem
{

public:

 /// Constructor:
 DiskShockWaveProblem();

 /// Run the problem; specify case_number to label output
 /// directory
 void run(const unsigned& case_number);

 /// Access function for the solid mesh
 ElasticRefineableQuarterCircleSectorMesh<ELEMENT>*& solid_mesh_pt()
  {
   return Solid_mesh_pt;
  }

 /// Access function for the mesh of surface traction elements
 SolidMesh*& traction_mesh_pt()
  {
   return Traction_mesh_pt;
  }
```

```
 /// Doc the solution
 void doc_solution();

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

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

 /// Actions after adaption: Kill and then re-build the traction
 /// elements on boundary 1 and re-assign the equation numbers
 void actions_after_adapt();

 /// Doc displacement and velocity: label file with before and after
 void doc_displ_and_veloc(const int& stage=0);

 /// Dump problem-specific parameters values, then dump
 /// generic problem data.
 void dump_it(ofstream& dump_file);

 /// Read problem-specific parameter values, then recover
 /// generic problem data.
 void restart(ifstream& restart_file);

private:

 // Output
 DocInfo Doc_info;

 /// Trace file
 ofstream Trace_file;

 /// Vector of pointers to nodes whose position we're tracing
 Vector<Node*> Trace_node_pt;

 /// Pointer to solid mesh
 ElasticRefineableQuarterCircleSectorMesh<ELEMENT>* Solid_mesh_pt;

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

};




//========================================================================
/// Constructor
//========================================================================
template<class ELEMENT, class TIMESTEPPER>
DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::DiskShockWaveProblem()
{


 //Allocate the timestepper
 add_time_stepper_pt(new TIMESTEPPER);
 // Set coordinates and radius for the circle that defines
 // the outer curvilinear boundary of the domain
 double x_c=0.0;
 double y_c=0.0;
 double r=1.0;
 // Build geometric object that specifies the fish back in the
 // undeformed configuration (basically a deep copy of the previous one)
 GeomObject* curved_boundary_pt=new Circle(x_c,y_c,r,time_stepper_pt());

 // The curved boundary of the mesh is defined by the geometric object
 // What follows are the start and end coordinates on the geometric object:
 double xi_lo=0.0;
 double xi_hi=2.0*atan(1.0);

 // Fraction along geometric object at which the radial dividing line
 // is placed
 double fract_mid=0.5;

 //Now create the mesh
 solid_mesh_pt() = new ElasticRefineableQuarterCircleSectorMesh<ELEMENT>(
  curved_boundary_pt,xi_lo,fract_mid,xi_hi,time_stepper_pt());

 // Set up trace nodes as the nodes on boundary 1 (=curved boundary) in
 // the original mesh (they exist under any refinement!)
 unsigned nnod0=solid_mesh_pt()->nboundary_node(0);
 unsigned nnod1=solid_mesh_pt()->nboundary_node(1);
 Trace_node_pt.resize(nnod0+nnod1);
 for (unsigned j=0;j<nnod0;j++)
  {
   Trace_node_pt[j]=solid_mesh_pt()->boundary_node_pt(0,j);
```

```
 }
for (unsigned j=0;j<nnod1;j++)
 {
  Trace_node_pt[j+nnod0]=solid_mesh_pt()->boundary_node_pt(1,j);
 }

// Build traction element mesh
solid_mesh_pt()->make_traction_element_mesh(traction_mesh_pt());
// Solid mesh is first sub-mesh
add_sub_mesh(solid_mesh_pt());

// Traction mesh is first sub-mesh
add_sub_mesh(traction_mesh_pt());

// Build combined "global" mesh
build_global_mesh();


// Create/set error estimator
solid_mesh_pt()->spatial_error_estimator_pt()=new Z2ErrorEstimator;

// Fiddle with adaptivity targets and doc
solid_mesh_pt()->max_permitted_error()=0.006; //0.03;
solid_mesh_pt()->min_permitted_error()=0.0006;// 0.0006; //0.003;
solid_mesh_pt()->doc_adaptivity_targets(cout);

// Pin the bottom in the vertical direction
unsigned n_bottom = solid_mesh_pt()->nboundary_node(0);

//Loop over the node
for(unsigned i=0;i<n_bottom;i++)
 {
  solid_mesh_pt()->boundary_node_pt(0,i)->pin_position(1);
 }

// Pin the left edge in the horizontal direction
unsigned n_side = solid_mesh_pt()->nboundary_node(2);
//Loop over the node
for(unsigned i=0;i<n_side;i++)
 {
  solid_mesh_pt()->boundary_node_pt(2,i)->pin_position(0);
 }

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

//Set parameters and function pointers for solid elements
for(unsigned i=0;i<n_element;i++)
 {
  //Cast to a solid element
  ELEMENT *el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));

  //Set the constitutive law
  el_pt->constitutive_law_pt() =
   Global_Physical_Variables::Constitutive_law_pt;

  // Switch on inertia
  el_pt->enable_inertia();
 }

// Pin the redundant solid pressures
PVDEquationsBase<2>::pin_redundant_nodal_solid_pressures(
 solid_mesh_pt()->element_pt());

//Find number of elements in traction mesh
n_element=traction_mesh_pt()->nelement();

//Set function pointers for traction elements
for(unsigned i=0;i<n_element;i++)
 {
  //Cast to a solid traction element
  SolidTractionElement<ELEMENT> *el_pt =
   dynamic_cast<SolidTractionElement<ELEMENT>*>
    (traction_mesh_pt()->element_pt(i));

  //Set the traction function
  el_pt->traction_fct_pt() = Global_Physical_Variables::constant_pressure;
 }

//Attach the boundary conditions to the mesh
cout << assign_eqn_numbers() << std::endl;

// Refine uniformly
refine_uniformly();
refine_uniformly();
refine_uniformly();
```

```
 // Now the non-pinned positions of the SolidNodes will have been
 // determined by interpolation. This is appropriate for uniform
 // refinements once the code is up and running since we can't place
 // new SolidNodes at the positions determined by the MacroElement.
 // However, here we want to update the nodes to fit the exact
 // intitial configuration.

 // Update all solid nodes based on the Mesh's Domain/MacroElement
 // representation
 bool update_all_solid_nodes=true;
 solid_mesh_pt()->node_update(update_all_solid_nodes);

 // Now set the Eulerian equal to the Lagrangian coordinates
 solid_mesh_pt()->set_lagrangian_nodal_coordinates();
}
```

```
//======================================================================
/// Kill and then re-build the traction elements on boundary 1,
/// pin redundant pressure dofs and re-assign the equation numbers.
//======================================================================
template<class ELEMENT, class TIMESTEPPER>
void DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::actions_after_adapt()
{
 // Wipe and re-build traction element mesh
 solid_mesh_pt()->remake_traction_element_mesh(traction_mesh_pt());
 // Re-build combined "global" mesh
 rebuild_global_mesh();

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

 //Loop over the elements in the traction element mesh
 for(unsigned i=0;i<n_element;i++)
  {
   //Cast to a solid traction element
   SolidTractionElement<ELEMENT> *el_pt =
    dynamic_cast<SolidTractionElement<ELEMENT>*>
    (traction_mesh_pt()->element_pt(i));

   //Set the traction function
   el_pt->traction_fct_pt() = Global_Physical_Variables::constant_pressure;
  }

 // Pin the redundant solid pressures
 PVDEquationsBase<2>::pin_redundant_nodal_solid_pressures(
  solid_mesh_pt()->element_pt());


 //Do equation numbering
 cout << assign_eqn_numbers() << std::endl;

}
```

```
//======================================================================
/// Doc the solution
//======================================================================
template<class ELEMENT, class TIMESTEPPER>
void DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::doc_solution()
{
 // Number of plot points
 unsigned npts;
 npts=5;

 // Output shape of deformed body
 ofstream some_file;
 char filename[100];
 sprintf(filename,"%s/soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 solid_mesh_pt()->output(some_file,npts);
 some_file.close();


 // Output traction
 unsigned nel=traction_mesh_pt()->nelement();
 sprintf(filename,"%s/traction%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Vector<double> unit_normal(2);
```

```
Vector<double> traction(2);
Vector<double> x_dummy(2);
Vector<double> s_dummy(1);
for (unsigned e=0;e<nel;e++)
 {
  some_file « "ZONE " « std::endl;
  for (unsigned i=0;i<npts;i++)
   {
    s_dummy[0]=-1.0+2.0*double(i)/double(npts-1);
    SolidTractionElement<ELEMENT>* el_pt=
     dynamic_cast<SolidTractionElement<ELEMENT>*>(
      traction_mesh_pt()->finite_element_pt(e));
    el_pt->outer_unit_normal(s_dummy,unit_normal);
    el_pt->traction(s_dummy,traction);
    el_pt->interpolated_x(s_dummy,x_dummy);
    some_file « x_dummy[0] « " " « x_dummy[1] « " "
              « traction[0] « " " « traction[1] « " "
              « std::endl;
   }
 }
some_file.close();

// Doc displacement and velocity
doc_displ_and_veloc();

// Get displacement as a function of the radial coordinate
// along boundary 0
{

 // Number of elements along boundary 0:
 unsigned nelem=solid_mesh_pt()->nboundary_element(0);

 // Open files
 sprintf(filename,"%s/displ%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);

 Vector<double> s(2);
 Vector<double> x(2);
 Vector<double> dxdt(2);
 Vector<double> xi(2);
 Vector<double> r_exact(2);
 Vector<double> v_exact(2);

 for (unsigned e=0;e<nelem;e++)
  {
   some_file « "ZONE " « std::endl;
   for (unsigned i=0;i<npts;i++)
    {
     // Move along bottom edge of element
     s[0]=-1.0+2.0*double(i)/double(npts-1);
     s[1]=-1.0;

     // Get pointer to element
     SolidFiniteElement* el_pt=dynamic_cast<SolidFiniteElement*>
      (solid_mesh_pt()->boundary_element_pt(0,e));

     // Get Lagrangian coordinate
     el_pt->interpolated_xi(s,xi);

     // Get Eulerian coordinate
     el_pt->interpolated_x(s,x);

     // Get velocity
     el_pt->interpolated_dxdt(s,1,dxdt);

     // Plot radial distance and displacement
     some_file « xi[0] « " " « x[0]-xi[0] « " "
               « dxdt[0] « std::endl;
    }
  }
 some_file.close();
}


// Write trace file
Trace_file « time_pt()->time()  « " ";
unsigned ntrace_node=Trace_node_pt.size();
for (unsigned j=0;j<ntrace_node;j++)
 {
  Trace_file « sqrt(pow(Trace_node_pt[j]->x(0),2)+
                    pow(Trace_node_pt[j]->x(1),2)) « " ";
 }
Trace_file « std::endl;


// removed until Jacobi eigensolver is re-instated
```

```
 // // Output principal stress vectors at the centre of all elements
 // SolidHelpers::doc_2D_principal_stress<ELEMENT>(Doc_info,solid_mesh_pt());

//  // Write restart file
//  sprintf(filename,"%s/restart%i.dat",Doc_info.directory().c_str(),
//          Doc_info.number());
//  some_file.open(filename);
//  dump_it(some_file);
//  some_file.close();

 cout « "Doced solution for step "
      « Doc_info.number()
      « std::endl « std::endl « std::endl;
}
```

```
//==================================================================
/// Doc displacement and veloc in displ_and_veloc*.dat.
/// The int stage defaults to 0, in which case the '*' in the
/// filename is simply the step number specified by the Problem's
/// DocInfo object. If it's +/-1, the word "before" and "after"
/// get inserted. This allows checking of the veloc/displacment
/// interpolation during adaptive mesh refinement.
//==================================================================
template<class ELEMENT, class TIMESTEPPER>
void DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::doc_displ_and_veloc(
 const int& stage)
{

 ofstream some_file;
 char filename[100];

 // Number of plot points
 unsigned npts;
 npts=5;

 // Open file
 if (stage==-1)
  {
   sprintf(filename,"%s/displ_and_veloc_before%i.dat",
           Doc_info.directory().c_str(),Doc_info.number());
  }
 else if (stage==1)
  {
   sprintf(filename,"%s/displ_and_veloc_after%i.dat",
           Doc_info.directory().c_str(),Doc_info.number());
  }
 else
  {
   sprintf(filename,"%s/displ_and_veloc%i.dat",
           Doc_info.directory().c_str(),Doc_info.number());
  }
 some_file.open(filename);

 Vector<double> s(2),x(2),dxdt(2),xi(2),displ(2);

 //Loop over solid elements
 unsigned nel=solid_mesh_pt()->nelement();
 for (unsigned e=0;e<nel;e++)
  {
   some_file « "ZONE I=" « npts « ", J=" « npts « std::endl;
   for (unsigned i=0;i<npts;i++)
    {
     s[0]=-1.0+2.0*double(i)/double(npts-1);
     for (unsigned j=0;j<npts;j++)
      {
       s[1]=-1.0+2.0*double(j)/double(npts-1);

       // Cast to full element type
       ELEMENT* el_pt=dynamic_cast<ELEMENT*>(solid_mesh_pt()->
                                              finite_element_pt(e));

       // Eulerian coordinate
       el_pt->interpolated_x(s,x);

       // Lagrangian coordinate
       el_pt->interpolated_xi(s,xi);

       // Displacement
       displ[0]=x[0]-xi[0];
       displ[1]=x[1]-xi[1];

       // Velocity (1st deriv)
```

```
       el_pt->interpolated_dxdt(s,1,dxdt);

       some_file « x[0] « " " « x[1] « " "
                 « displ[0] « " " « displ[1] « " "
                 « dxdt[0] « " " « dxdt[1] « " "
                 « std::endl;
      }
    }
  }
 some_file.close();

}



//========================================================================
/// Dump the solution
//========================================================================
template<class ELEMENT, class TIMESTEPPER>
void DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::dump_it(ofstream& dump_file)
{
 // Call generic dump()
 Problem::dump(dump_file);
}



//========================================================================
/// Read solution from disk
//========================================================================
template<class ELEMENT, class TIMESTEPPER>
void DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::restart(ifstream& restart_file)
{
 // Read generic problem data
 Problem::read(restart_file);
}



//==================================================================
/// Run the problem; specify case_number to label output directory
//==================================================================
template<class ELEMENT, class TIMESTEPPER>
void DiskShockWaveProblem<ELEMENT,TIMESTEPPER>::run(
 const unsigned& case_number)
{

 // If there's a command line argument, run the validation (i.e. do only
 // 3 timesteps; otherwise do a few cycles
 unsigned nstep=400;
 if (CommandLineArgs::Argc!=1)
  {
   nstep=3;
  }

 // Define output directory
 char dirname[100];
 sprintf(dirname,"RESLT%i",case_number);
 Doc_info.set_directory(dirname);

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

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

 // Set up trace nodes as the nodes on boundary 1 (=curved boundary) in
 // the original mesh (they exist under any refinement!)
 unsigned nnod0=solid_mesh_pt()->nboundary_node(0);
 unsigned nnod1=solid_mesh_pt()->nboundary_node(1);
 Trace_file « "VARIABLES=\"time\"";
 for (unsigned j=0;j<nnod0;j++)
  {
   Trace_file « ", \"radial node " « j « "\" ";
  }
 for (unsigned j=0;j<nnod1;j++)
  {
   Trace_file « ", \"azimuthal node " « j « "\" ";
  }
 Trace_file « std::endl;


//  // Restart?
```

```
//  //---------

//  // Pointer to restart file
//  ifstream* restart_file_pt=0;

//  // No restart
//  //-----------
//  if (CommandLineArgs::Argc==1)
//   {
//    cout « "No restart" « std::endl;
//   }
//  // Restart
//  //--------
//  else if (CommandLineArgs::Argc==2)
//   {
//    // Open restart file
//    restart_file_pt=new ifstream(CommandLineArgs::Argv[1],ios_base::in);
//    if (restart_file_pt!=0)
//     {
//      cout « "Have opened " « CommandLineArgs::Argv[1] «
//       " for restart. " « std::endl;
//     }
//    else
//     {
//      cout « "ERROR while trying to open " « CommandLineArgs::Argv[1] «
//       " for restart." « std::endl;
//     }
//    // Do the actual restart
//    pause("need to do the actual restart");
//    //problem.restart(*restart_file_pt);
//   }
//  // More than one restart file specified?
//  else
//   {
//    cout « "Can only specify one input file " « std::endl;
//    cout « "You specified the following command line arguments: " « std::endl;
//    CommandLineArgs::output();
//    //assert(false);
//   }


 // Initial parameter values
 Global_Physical_Variables::P = 0.1;

 // Initialise time
 double time0=0.0;
 time_pt()->time()=time0;

 // Set initial timestep
 double dt=0.01;

 // Impulsive start
 assign_initial_values_impulsive(dt);
 // Doc initial state
 doc_solution();
 Doc_info.number()++;

 // First step without adaptivity
 unsteady_newton_solve(dt);
 doc_solution();
 Doc_info.number()++;

 //Timestepping loop for subsequent steps with adaptivity
 unsigned max_adapt=1;
 for(unsigned i=1;i<nstep;i++)
  {
   unsteady_newton_solve(dt,max_adapt,false);
   doc_solution();
   Doc_info.number()++;
  }

}





//======================================================================
/// Driver for simple elastic problem
//======================================================================
int main(int argc, char* argv[])
{

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

 //Initialise physical parameters
```

```
Global_Physical_Variables::E = 1.0; // ADJUST
Global_Physical_Variables::Nu = 0.3; // ADJUST

 // "Big G" Linear constitutive equations:
 Global_Physical_Variables::Constitutive_law_pt =
  new GeneralisedHookean(&Global_Physical_Variables::Nu,
                         &Global_Physical_Variables::E);
 //Set up the problem:
 unsigned case_number=0;


// Pure displacement formulation
 {
  cout « "Running case " « case_number
       « ": Pure displacement formulation" « std::endl;
  DiskShockWaveProblem<RefineableQPVDElement<2,3>, Newmark<1> > problem;
  problem.run(case_number);
  case_number++;
 }

// Pressure-displacement with Crouzeix Raviart-type pressure
 {
  cout « "Running case " « case_number
       « ": Pressure/displacement with Crouzeix-Raviart pressure" « std::endl;
  DiskShockWaveProblem<RefineableQPVDElementWithPressure<2>, Newmark<1> >
   problem;
  problem.run(case_number);
  case_number++;
 }


 // Pressure-displacement with Taylor-Hood-type pressure
 {
  cout « "Running case " « case_number
       « ": Pressure/displacement with Taylor-Hood pressure" « std::endl;
  DiskShockWaveProblem<RefineableQPVDElementWithContinuousPressure<2>,
   Newmark<1> > problem;
  problem.run(case_number);
  case_number++;
 }

// Clean up
delete Global_Physical_Variables::Constitutive_law_pt;
Global_Physical_Variables::Constitutive_law_pt=0;

}
```

## 1.1 PDF file

A `pdf version` of this document is available.