

Chapter 1

Demo problem: Large-amplitude shear deformation of a 3D elastic solid

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 elastic deformation of a cuboidal domain
// The deformation is a simple shear in the x-z plane driven by
// motion of the top boundary, for exact solution see Green & Zerna

// Generic oomph-lib headers
#include "generic.h"
// Solid mechanics
#include "solid.h"
// The mesh
#include "meshes/simple_cubic_mesh.template.h"
using namespace std;
using namespace oomph;

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

//=====
/// Simple cubic mesh upgraded to become a solid mesh
//=====
template<class ELEMENT>
class ElasticCubicMesh : public virtual SimpleCubicMesh<ELEMENT>,
                        public virtual SolidMesh
{
public:

    /// \short Constructor:
```

```

ElasticCubicMesh(const unsigned &nx, const unsigned &ny, const unsigned &nz,
                 const double &a, const double &b, const double &c,
                 TimeStepper* time_stepper_pt=&Mesh::Default_TimeStepper) :
SimpleCubicMesh<ELEMENT>(nx,ny,nz,-a,a,-b,b,-c,c,time_stepper_pt)
{
    //Assign the initial lagrangian coordinates
    set_lagrangian_nodal_coordinates();
}

/// Empty Destructor
virtual ~ElasticCubicMesh() { }
};

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

//=====
/// Global variables
//=====
namespace Global_Physical_Variables
{
    /// Pointer to strain energy function
    StrainEnergyFunction* Strain_energy_function_pt;

    /// Pointer to constitutive law
    ConstitutiveLaw* Constitutive_law_pt;

    /// Elastic modulus
    double E=1.0;

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

    /// "Mooney Rivlin" coefficient for generalised Mooney Rivlin law
    double C1=1.3;

    /// Body force
    double Gravity=0.0;

    /// Body force vector: Vertically downwards with magnitude Gravity
    void body_force(const Vector<double>& xi,
                   const double& t,
                   Vector<double>& b)
    {
        b[0]=0.0;
        b[1]=-Gravity;
    }
}

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

//=====
/// Boundary-driven elastic deformation of fish-shaped domain.
//=====
template<class ELEMENT>
class SimpleShearProblem : public Problem
{
    double Shear;
    void set_incompressible(ELEMENT *el_pt,const bool &incompressible);
public:

    /// Constructor:
    SimpleShearProblem(const bool &incompressible);

    /// Run simulation.
    void run(const std::string &dirname);

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

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

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

    /// \short Update before solve: We're dealing with a static problem so
    /// the nodal positions before the next solve merely serve as
    /// initial conditions. For meshes that are very strongly refined
    /// near the boundary, the update of the displacement boundary
    /// conditions (which only moves the SolidNodes *on* the boundary),
    /// can lead to strongly distorted meshes. This can cause the
    /// Newton method to fail --> the overall method is actually more robust

```

```

/// if we use the nodal positions as determined by the Domain/MacroElement-
/// based mesh update as initial guesses.
void actions_before_newton_solve()
{
    apply_boundary_conditions();
    bool update_all_solid_nodes=true;
    mesh_pt()->node_update(update_all_solid_nodes);
}

///Shear the top
void apply_boundary_conditions()
{
    unsigned ibound = 5;
    unsigned num_nod=mesh_pt()->nboundary_node(ibound);
    for (unsigned inod=0; inod<num_nod; inod++)
    {
        SolidNode *solid_nod_pt = static_cast<SolidNode*>(
            mesh_pt()->boundary_node_pt(ibound,inod));
        solid_nod_pt->x(0) = solid_nod_pt->xi(0) + Shear*
            solid_nod_pt->xi(2);
    }
};

//=====
/// Constructor:
//=====
template<class ELEMENT>
SimpleShearProblem<ELEMENT>::SimpleShearProblem(const bool &incompressible)
: Shear(0.0)
{
    double a = 1.0, b = 1.0, c = 1.0;
    unsigned nx = 5, ny = 5, nz = 5;
    // Build mesh
    Problem::mesh_pt()=new ElasticCubicMesh<ELEMENT>(nx,ny,nz,a,b,c);

    //Loop over all boundaries
    for(unsigned b=0;b<6;b++)
    {
        //Loop over nodes in the boundary
        unsigned n_node = mesh_pt()->nboundary_node(b);
        for(unsigned n=0;n<n_node;n++)
        {
            //Pin all nodes in the y and z directions to keep the motion in plane
            for(unsigned i=1;i<3;i++)
            {
                mesh_pt()->boundary_node_pt(b,n)->pin_position(i);
            }
            //On the top and bottom pin the positions in x
            if((b==0) || (b==5))
            {
                mesh_pt()->boundary_node_pt(b,n)->pin_position(0);
            }
        }
    }

    //Loop over the elements in the mesh to set parameters/function pointers
    unsigned n_element =mesh_pt()->nelement();
    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() =
            GlobalPhysicalVariables::Constitutive_law_pt;
        set_incompressible(el_pt,incompressible);

        // Set the body force
        //el_pt->body_force_fct_pt()=GlobalPhysicalVariables::body_force;
    }

    // Pin the redundant solid pressures (if any)
    //PVDEquationsBase<2>::pin_redundant_nodal_solid_pressures(
    // mesh_pt()->element_pt());
    //Attach the boundary conditions to the mesh
    cout << assign_eqn_numbers() << std::endl;
}

//=====
/// Doc the solution
//=====
template<class ELEMENT>
void SimpleShearProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
    ofstream some_file;
    char filename[100];
    // Number of plot points
    unsigned npts = 5;

```

```

// Output shape of deformed body
sprintf(filename, "%s/soln%i.dat", doc_info.directory().c_str(),
        doc_info.number());
some_file.open(filename);
mesh_pt()->output(some_file, npts);
some_file.close();

sprintf(filename, "%s/stress%i.dat", doc_info.directory().c_str(),
        doc_info.number());
some_file.open(filename);
//Output the appropriate stress at the centre of each element
Vector<double> s(3,0.0);
Vector<double> x(3);
DenseMatrix<double> sigma(3,3);

unsigned n_element = mesh_pt()->nelement();
for(unsigned e=0;e<n_element;e++)
{
    ELEMENT* el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e));
    el_pt->interpolated_x(s,x);
    el_pt->get_stress(s,sigma);
    //Output
    for(unsigned i=0;i<3;i++)
    {
        some_file << x[i] << " ";
    }
    for(unsigned i=0;i<3;i++)
    {
        for(unsigned j=0;j<3;j++)
        {
            some_file << sigma(i,j) << " ";
        }
    }
    some_file << std::endl;
}
some_file.close();
}

//=====
// Run the problem
//=====
template<class ELEMENT>
void SimpleShearProblem<ELEMENT>::run(const std::string &dirname)
{
    // Output
    DocInfo doc_info;

    // Set output directory
    doc_info.set_directory(dirname);
    // Step number
    doc_info.number()=0;

    // Initial parameter values

    // Gravity:
    Global_Physical_Variables::Gravity=0.1;

    //Parameter incrementation
    unsigned nstep=2;
    for(unsigned i=0;i<nstep;i++)
    {
        //Solve the problem with Newton's method, allowing for up to 5
        //rounds of adaptation
        newton_solve();
        // Doc solution
        doc_solution(doc_info);
        doc_info.number()++;
        //Increase the shear
        Shear += 0.5;
    }
}

template<>
void SimpleShearProblem<QPVElement<3,3>>::set_incompressible(
    QPVElement<3,3> *el_pt, const bool &incompressible)
{
    //Does nothing
}

template<>
void SimpleShearProblem<QPVElementWithPressure<3>>::set_incompressible(
    QPVElementWithPressure<3> *el_pt, const bool &incompressible)
{
    if(incompressible) {el_pt->set_incompressible();}
    else {el_pt->set_compressible();}
}

template<>
void SimpleShearProblem<QPVElementWithContinuousPressure<3>>::
set_incompressible(

```

```

QPVDElementWithContinuousPressure<3> *el_pt, const bool &incompressible)
{
    if(incompressible) {el_pt->set_incompressible();}
    else {el_pt->set_compressible();}
}
//=====
/// Driver for simple elastic problem
//=====
int main()
{
    //Initialise physical parameters
    Global_Physical_Variables::E = 2.1;
    Global_Physical_Variables::Nu = 0.4;
    Global_Physical_Variables::C1 = 1.3;

    for (unsigned i=0;i<2;i++)
    {
        // Define a strain energy function: Generalised Mooney Rivlin
        Global_Physical_Variables::Strain_energy_function_pt =
            new GeneralisedMooneyRivlin(&Global_Physical_Variables::Nu,
                                        &Global_Physical_Variables::C1,
                                        &Global_Physical_Variables::E);

        // Define a constitutive law (based on strain energy function)
        Global_Physical_Variables::Constitutive_law_pt =
            new IsotropicStrainEnergyFunctionConstitutiveLaw(
                Global_Physical_Variables::Strain_energy_function_pt);
        {
            //Set up the problem with pure displacement formulation
            SimpleShearProblem<QPVDElement<3,3> > problem(false);
            problem.run("RESLT");
        }
        //Discontinuous pressure
        {
            //Set up the problem with pure displacement formulation
            SimpleShearProblem<QPVDElementWithPressure<3> > problem(false);
            problem.run("RESLT_pres");
        }
        /*{
            //Set up the problem with pure displacement formulation
            SimpleShearProblem<QPVDElementWithPressure<3> > problem(true);
            problem.run("RESLT_pres_incomp");
        }*/
        {
            //Set up the problem with pure displacement formulation
            SimpleShearProblem<QPVDElementWithContinuousPressure<3> > problem(false);
            problem.run("RESLT_cont_pres");
        }
        /*{
            //Set up the problem with pure displacement formulation
            SimpleShearProblem<QPVDElementWithContinuousPressure<3> > problem(true);
            problem.run("RESLT_cont_pres_incomp");
        }*/
    }
}
}

```

1.1 PDF file

A [pdf version](#) of this document is available.