

Chapter 1

Demo problem: Buckling of a clamped cylindrical shell under pressure loading

In this document, we discuss the solution of the buckling of a cylindrical shell using oomph-lib's Kirchhoff-LoveShell elements.

[No documentation yet: Here's the 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) 2006-2025 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 function for a simple test shell problem:
//Calculate the deformation of an elastic tube approximated
//using Kirchoff--Love shell theory

//Standard system includes
#include <iostream>
#include <fstream>
#include <cmath>
#include <typeinfo>
#include <algorithm>
#include <cstdio>

//Include files from the finite-element library
#include "generic.h"
#include "shell.h"
#include "meshes/rectangular_quadmesh.h"

using namespace std;

using namespace oomph;

//=====
/// Global variables that represent physical properties
//=====
namespace Global_Physical_Variables
{
```

```

/// Prescribed position of control point
double Prescribed_y = 1.0;

/// Pointer to pressure load (stored in Data so it can
/// become an unknown in the problem when displacement control is used
Data* Pext_data_pt;

/// Perturbation pressure
double Pcos=1.0;

/// Return a reference to the external pressure
/// load on the elastic tube.
double external_pressure()
{return (*Pext_data_pt->value_pt(0))*pow(0.05,3)/12.0; }

/// Load function, normal pressure loading
void press_load(const Vector<double> &xi,
                const Vector<double> &x,
                const Vector<double> &N,
                Vector<double>& load)
{
//std::cout << N[0] << " " << N[1] << " " << N[2] << std::endl;
//std::cout << xi[0] << " " << xi[1] << std::endl;
for(unsigned i=0;i<3;i++)
{
    load[i] = (external_pressure() -
               Pcos*pow(0.05,3)/12.0*cos(2.0*xi[1]))*N[i];
}
}

//=====================================================================
/// A 2D Mesh class. The tube wall is represented by two Lagrangian
/// coordinates that correspond to z and theta in cylindrical polars.
/// The required mesh is therefore a 2D mesh and is therefore inherited
/// from the generic RectangularQuadMesh
//=====================================================================
template <class ELEMENT>
class ShellMesh : public virtual RectangularQuadMesh<ELEMENT>,
                  public virtual SolidMesh
{
public:

    /// Constructor for the mesh
    ShellMesh(const unsigned &nx, const unsigned &ny,
              const double &lx, const double &ly);

    /// In all elastic problems, the nodes must be assigned an undeformed,
    /// or reference, position, corresponding to the stress-free state
    /// of the elastic body. This function assigns the undeformed position
    /// for the nodes on the elastic tube
    void assign_undeformed_positions(GeomObject* const &undeformed_midplane_pt);

};

//=====================================================================
/// Mesh constructor
/// Argument list:
/// nx : number of elements in the axial direction
/// ny : number of elements in the azimuthal direction
/// lx : length in the axial direction
/// ly : length in theta direction
//=====================================================================
template <class ELEMENT>
ShellMesh<ELEMENT>::ShellMesh(const unsigned &nx,
                               const unsigned &ny,
                               const double &lx,
                               const double &ly) :
RectangularQuadMesh<ELEMENT>(nx,ny,lx,ly)
{
    //Find out how many nodes there are
    unsigned n_node = nnodes();
    //Now in this case it is the Lagrangian coordinates that we want to set,
    //so we have to loop over all nodes and set them to the Eulerian
    //coordinates that are set by the generic mesh generator
    for(unsigned i=0;i<n_node;i++)
    {
        node_pt(i)->xi(0) = node_pt(i)->x(0);
        node_pt(i)->xi(1) = node_pt(i)->x(1);
    }
}

```

```

//Assign gradients, etc for the Lagrangian coordinates of
//hermite-type elements
//Read out number of position dofs
unsigned n_position_type = finite_element_pt(0)->nnodal_position_type();

//If this is greater than 1 set the slopes, which are the distances between
//nodes. If the spacing were non-uniform, this part would be more difficult
if(n_position_type > 1)
{
    double xstep = (this->Xmax - this->Xmin)/((this->Np-1)*this->Nx);
    double ystep = (this->Ymax - this->Ymin)/((this->Np-1)*this->Ny);
    for(unsigned n=0;n<n_node;n++)
    {
        //The factor 0.5 is because our reference element has length 2.0
        node_pt(n)->xi_gen(1,0) = 0.5*xstep;
        node_pt(n)->xi_gen(2,1) = 0.5*ystep;
    }
}
}

//=====================================================================
/// Set the undeformed coordinates of the nodes
//=====================================================================
template <class ELEMENT>
void ShellMesh<ELEMENT>::assign_undeformed_positions(
    GeomObject* const &undeformed_midplane_pt)
{
    //Find out how many nodes there are
    unsigned n_node = nnode();

    //Loop over all the nodes
    for(unsigned n=0;n<n_node;n++)
    {
        //Get the Lagrangian coordinates
        Vector<double> xi(2);
        xi[0] = node_pt(n)->xi(0);
        xi[1] = node_pt(n)->xi(1);

        //Assign memory for values of derivatives, etc
        Vector<double> R(3);
        DenseMatrix<double> a(2,3);
        RankThreeTensor<double> dadxi(2,2,3);

        //Get the geometrical information from the geometric object
        undeformed_midplane_pt->d2position(xi,R,a,dadxi);

        //Loop over coordinate directions
        for(unsigned i=0;i<3;i++)
        {
            //Set the position
            node_pt(n)->x_gen(0,i) = R[i];

            //Set the derivative wrt Lagrangian coordinates
            //Note that we need to scale by the length of each element here!!
            node_pt(n)->x_gen(1,i) = 0.5*a(0,i)*((this->Xmax - this->Xmin)/this->Nx);
            node_pt(n)->x_gen(2,i) = 0.5*a(1,i)*((this->Ymax - this->Ymin)/this->Ny);

            //Set the mixed derivative
            //symmetric so doesn't matter which one we use
            node_pt(n)->x_gen(3,i) = 0.25*dadxi(0,1,i);
        }
    }
}

//=====================================================================
//Problem class to solve the deformation of an elastic tube
//=====================================================================
template<class ELEMENT>
class ShellProblem : public Problem
{
public:

    /// Constructor
    ShellProblem(const unsigned &nx, const unsigned &ny,
                 const double &lx, const double &ly);

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

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

```

```

/// Actions before solve empty
void actions_before_newton_solve() {}
//A self_test function
void solve();

private:

    /// Pointer to GeomObject that specifies the undeformed midplane
GeomObject* Undeformed_midplane_pt;

    /// First trace node
Node* Trace_node_pt;

    /// Second trace node
Node* Trace_node2_pt;

};

//=====================================================================
// Constructor
//=====================================================================
template<class ELEMENT>
ShellProblem<ELEMENT>::ShellProblem(const unsigned &nx, const unsigned &ny,
                                      const double &lx, const double &ly)
{
    //Create the undeformed midplane object
    Undeformed_midplane_pt = new EllipticalTube(1.0,1.0);

    //Now create the mesh
    Problem::mesh_pt() = new ShellMesh<ELEMENT>(nx,ny,lx,ly);

    //Set the undeformed positions in the mesh
    mesh_pt() ->assign_undeformed_positions(Undeformed_midplane_pt);

    //Reorder the elements, since I know what's best for them....
    mesh_pt() ->element_reorder();

    //Apply boundary conditions to the ends of the tube
    unsigned n_ends = mesh_pt() ->nboundary_node(1);
    //Loop over the node
    for(unsigned i=0;i<n_ends;i++)
    {
        //Pin in the axial direction (prevents rigid body motions)
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(2);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(2);
        //Derived conditions
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(2,2);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(2,2);

        //-----CLAMPING CONDITIONS-----
        //----Pin positions in the transverse directions-----
        // Comment these out to get the ring case
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(0);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(0);
        //Derived conditions
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(2,0);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(2,0);

        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(1);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(1);
        //Derived conditions
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(2,1);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(2,1);
        //-----

        // Set the axial gradients of the transverse coordinates to be
        // zero --- need to be enforced for ring or tube buckling
        //Pin dx/dz and dy/dz
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(1,0);
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(1,1);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(1,0);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(1,1);
        //Derived conditions
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(3,0);
        mesh_pt() ->boundary_node_pt(1,i) ->pin_position(3,1);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(3,0);
        mesh_pt() ->boundary_node_pt(3,i) ->pin_position(3,1);
    }

    //Now loop over the sides and apply symmetry conditions
    unsigned n_side = mesh_pt() ->nboundary_node(0);
    for(unsigned i=0;i<n_side;i++)
    {
        //At the side where theta is 0, pin in the y direction

```

```

mesh_pt() ->boundary_node_pt(0,i) ->pin_position(1);
//Derived condition
mesh_pt() ->boundary_node_pt(0,i) ->pin_position(1,1);
//Pin dx/dtheta and dz/dtheta
mesh_pt() ->boundary_node_pt(0,i) ->pin_position(2,0);
mesh_pt() ->boundary_node_pt(0,i) ->pin_position(2,2);
//Pin in the mixed derivative
mesh_pt() ->boundary_node_pt(0,i) ->pin_position(3,0);
mesh_pt() ->boundary_node_pt(0,i) ->pin_position(3,2);

//At the side when theta is 0.5pi pin in the x direction
mesh_pt() ->boundary_node_pt(2,i) ->pin_position(0);
//Derived condition
mesh_pt() ->boundary_node_pt(2,i) ->pin_position(1,0);
//Pin dy/dtheta and dz/dtheta
mesh_pt() ->boundary_node_pt(2,i) ->pin_position(2,1);
mesh_pt() ->boundary_node_pt(2,i) ->pin_position(2,2);
//Pin in the mixed derivative
mesh_pt() ->boundary_node_pt(2,i) ->pin_position(3,1);
mesh_pt() ->boundary_node_pt(2,i) ->pin_position(3,2);

//      //Set an initial kick to make sure that we hop onto the
//      //non-axisymmetric branch
//      if((i>1) && (i<n_side-1))
//      {
//          mesh_pt() ->boundary_node_pt(0,i) ->x(0) += 0.05;
//          mesh_pt() ->boundary_node_pt(2,i) ->x(1) -= 0.1;
//      }
}

// Setup displacement control
//-----

// //Setup displacement control
// //Fix the displacement at the mid-point of the tube in the "vertical"
// //((y) direction.
// //Set the displacement control element (located halfway along the tube)
// Disp_ctl_element_pt = dynamic_cast<ELEMENT*>(mesh_pt() ->element_pt(3*Ny-1));
// //The midpoint of the tube is located exactly half-way along the element
// Vector<double> s(2); s[0] = 1.0; s[1] = 0.0; //s[1] = 0.5
// //Fix the displacement at this point in the y (1) direction
// Disp_ctl_element_pt->fix_displacement_for_displacement_control(s,1);
// //Set the pointer to the prescribed position
// Disp_ctl_element_pt->prescribed_position_pt() = &Prescribed_y;

// Choose element in which displacement control is applied: This
// one is located about halfway along the tube -- remember that
// we've renumbered the elements!
unsigned nel_ctrl=0;
Vector<double> s_displ_control(2);

// Even/odd number of elements in axial direction
if (nx%2==1)
{
    nel_ctrl=unsigned(floor(0.5*double(nx))+1.0)*ny-1;
    s_displ_control[0]=0.0;
    s_displ_control[1]=1.0;
}
else
{
    nel_ctrl=unsigned(floor(0.5*double(nx))+1.0)*ny-1;
    s_displ_control[0]=-1.0;
    s_displ_control[1]=1.0;
}

// Controlled element
SolidFiniteElement* controlled_element_pt=
    dynamic_cast<ELEMENT*>(mesh_pt() ->element_pt(nel_ctrl));

// Fix the displacement in the y (1) direction...
unsigned controlled_direction=1;

// Pointer to displacement control element
DisplacementControlElement* displ_control_el_pt;
// Build displacement control element
displ_control_el_pt=
    new DisplacementControlElement(controlled_element_pt,
                                    s_displ_control,
                                    controlled_direction,
                                    &Global_Physical_Variables::Prescribed_y);

// Doc control point
Vector<double> xi(2);

```

```

Vector<double> x(3);
controlled_element_pt->interpolated_xi(s_displ_control,xi);
controlled_element_pt->interpolated_x(s_displ_control,x);
std::cout << std::endl;
std::cout << "Controlled element: " << nel_ctrl << std::endl;
std::cout << "Displacement control applied at xi = (" 
    << xi[0] << ", " << xi[1] << ")" << std::endl;
std::cout << "Corresponding to           x = (" 
    << x[0] << ", " << x[1] << ", " << x[2] << ")" << std::endl;

// The constructor of the DisplacementControlElement has created
// a new Data object whose one-and-only value contains the
// adjustable load: Use this Data object in the load function:
Global_Physical_Variables::Pext_data_pt=displ_control_el_pt->
    displacement_control_load_pt();
// Add the displacement-control element to the mesh
mesh_pt()->add_element_pt(displ_control_el_pt);

// Complete build of shell elements
//-----
//Find number of shell elements in mesh
unsigned n_element = nx*ny;

//Explicit pointer to first element in the mesh
ELEMENT* first_el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(0));
//Loop over the elements
for(unsigned e=0;e<n_element;e++)
{
    //Cast to a shell element
    ELEMENT *el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e));

    //Set the load function
    el_pt->load_vector_fct_pt() = & Global_Physical_Variables::press_load;

    //Set the undeformed surface
    el_pt->undeformed_midplane_pt() = Undeformed_midplane_pt;

    //The external pressure is external data for all elements
    el_pt->add_external_data(Global_Physical_Variables::Pext_data_pt);

    //Pre-compute the second derivatives wrt Lagrangian coordinates
    //for the first element only
    if(e==0)
    {
        el_pt->pre_compute_d2shape_lagrangian_at_knots();
    }

    //Otherwise set the values to be the same as those in the first element
    //this is OK because the Lagrangian mesh is uniform.
    else
    {
        el_pt->set_dshape_lagrangian_stored_from_element(first_el_pt);
    }
}

//Set pointers to two trace nodes, used for output
Trace_node_pt = mesh_pt()->finite_element_pt(2*ny-1)->node_pt(3);
Trace_node2_pt = mesh_pt()->finite_element_pt(ny)->node_pt(1);

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

//=====
// Define the solve function, disp ctrl and then continuation
//=====
template<class ELEMENT>
void ShellProblem<ELEMENT>::solve()
{
    //Increase the maximum number of Newton iterations.
    //Finding the first buckled solution requires a large(ish) number
    //of Newton steps -- shells are just a bit twitchy
    Max_newton_iterations = 40;
    Max_residuals=1.0e6;

    //Open an output trace file
}

```

```
ofstream trace("trace.dat");

//Gradually compress the tube by decreasing the value of the prescribed
//position
for(unsigned i=1;i<11;i++)
{
    Global_Physical_Variables::Prescribed_y -= 0.05;

    cout << std::endl << "Increasing displacement: Prescribed_y is "
    << Global_Physical_Variables::Prescribed_y << std::endl;

    // Solve
    newton_solve();

    //Output the pressure (on the bending scale)
    trace << Global_Physical_Variables::external_pressure() / (pow(0.05,3)/12.0)
    << " "
    //Position of first trace node
    << Trace_node_pt->x(0) << " " << Trace_node_pt->x(1) << " "
    //Position of second trace node
    << Trace_node2_pt->x(0) << " " << Trace_node2_pt->x(1) << std::endl;

    // Reset perturbation
    Global_Physical_Variables::Pcos=0.0;
}

//Close the trace file
trace.close();
//Output the tube shape in the most strongly collapsed configuration
ofstream file("final_shape.dat");
mesh_pt()->output(file,5);
file.close();

}

//=====================================================================
/// Driver
//=====================================================================
int main()
{
    //Length of domain
    double L = 10.0;
    double L_phi=0.5*MathematicalConstants::Pi;

    //Set up the problem
    ShellProblem<StorableShapeSolidElement<DiagHermiteShellElement> >
    problem(5,3,L,L_phi);

    //Solve the problem
    problem.solve();
}
```

1.1 PDF file

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