Chapter 1

The deformation of a thin-shell material with a small strain, using the Kirchhoff-Love shell theory

In this document, we discuss the solution of a simple two-dimensional thin elastic problem, using oomph-lib's Kirchhoff-Love shell elements.

Specifically, in this document we demonstrate

- the descriptions of the governing equation of a thin-shell deformation with a small strain

and

- how to implement a shell problem with the `BellElement` and the `C1CurvedElement` in `oomph-lib`

The reader is referred to the Bell triangular finite element and the C^1-curved triangular finite element tutorials, for more detailed descriptions on the `BellElement` and `C1Curved↩Element`.

1.1 Overview of a thin shell

A shell is defined as a thin three-dimensional elastic body where the thickness, h, is smaller compared to the other two dimensions. Many analyses of thin shells neglect the effect of the transverse shear and follow the theory of Kirchhoff-Love. This theory states that a normal vector of the undeformed mid-surface remains normal to the deformed mid-surface throughout the deformation and it deforms inextensionally.

Employing the Kirchhoff-Love assumption in a shell theory intends to reduce the dimension of a shell problem from the three-dimensional to the two-dimensional theory. Therefore, the shell governing equation which is derived from the principle of virtual displacement can be reduced to two-dimensional space. This is a result of allowing the integration in the coordinate perpendicular to the mid-surface to be carried out analytically. Therefore, all quantities can be expressed only on the two Lagrangian coordinates of the mid-surface.

Since the elastic material under consideration is a thin shell, the structure can experience large deflections and rotations, although strains and stresses may remain small. In such thin bodies, an assumption for small deformations (strains) is employed to simplify excessive deflections. This has been done by employing the linear (or Hookean) constitutive equation to represent the stress components as a linear function of all strain components.

1.1.1 A thin-shell governing equation with a small strain

In this section, we illustrate the linearisation of the governing equation of a static shell in a general geometry. It will be seen that a thin-elastic shell can be modelled by equations defined on a two-dimensional domain.

To develop a mathematical description of the deformation of a three-dimensional shell, we consider a shell geometry illustrated in the following figure. From the thinness feature of a shell, its geometry is allowed to specify by the two-dimensional reference surface and its thickness. We can choose the coordinate $\xi_3 = 0$ to be the shell's mid-surface and the reference surface. Therefore, we have that two coordinates ξ_1 and ξ_2 are located on the reference surface where the third coordinate ξ_3 is normal to the reference surface. The upper and lower surfaces of the shell have the coordinates $\xi_3 = h/2$ and $\xi_3 = -h/2$, respectively, where h is the thickness of the shell.

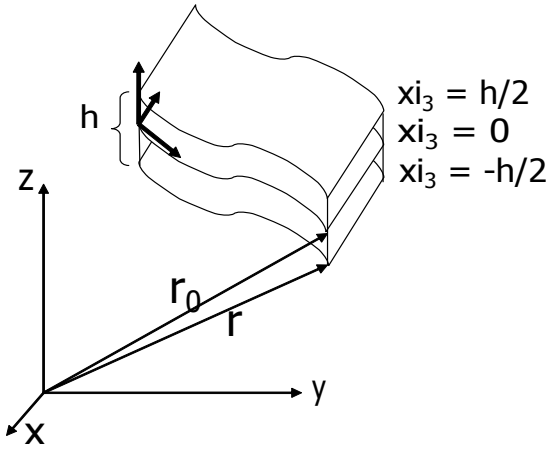

Figure 1.1 A shell geometry.

The governing equation of a static shell with zero pre-stress is parametrised by the two coordinates ξ_1, ξ_2 on the mid-surface of the shell and can be obtained as

$$\int_\Omega \left\{ \tilde{E}^{\alpha\beta\gamma\delta} \left(\gamma_{\alpha\beta}\delta\gamma_{\gamma\delta} + \frac{1}{12}h^2\kappa_{\alpha\beta}\delta\kappa_{\gamma\delta} \right) - \frac{1}{h}\sqrt{\frac{A}{a}}\mathbf{f}\cdot\delta\mathbf{R} \right\} \sqrt{a}d\xi_1 d\xi_2 = 0, \quad (1)$$

where $\tilde{E}^{\alpha\beta\gamma\delta}$ represents the plane stress stiffness tensor defined on the mid-surface and can be computed by

$$\tilde{E}^{\alpha\beta\gamma\delta} = \frac{E}{2(1+\nu)} \left(t^{\alpha\gamma}t^{\beta\delta} + t^{\alpha\delta}t^{\beta\gamma} + \frac{2\nu}{1-\nu}t^{\alpha\beta}t^{\gamma\delta} \right), \quad (2)$$

where ν denotes the Poisson's ratio. This equation is presented with a non-dimensional form and is described in details in another tutorial. Non-dimensionalisation of quantities is also explained here.

Note that in the small strain regime, an area element between the undeformed and deformed configurations are indistinguishable. Hence, the external force is preferable to express on the area element of the undeformed midplane which is related with that of the deformed midplane as

$$\hat{\mathbf{f}}\sqrt{a} = \mathbf{f}\sqrt{A},$$

where $\mathbf{f}, \hat{\mathbf{f}}$ denote the external force expressed on the area element of the undeformed and the deformed midplane, respectively.

In order to obtain a linearised version of the governing equation of a thin-shell deformation, all terms in the equation (1) have to be linearised. A small strain is then assumed. Therefore, we have that $\forall\epsilon << 1$,

$$\mathbf{f} = \epsilon\tilde{\mathbf{f}}, \quad \mathbf{u} = \epsilon\tilde{\mathbf{u}},$$

where a displacement $\mathbf{u}$ will be considered in the tangential and normal directions (rather than the Cartesian system) in this study. Therefore, a displacement $\mathbf{u}$ on the midplane which is parametrised by the two-dimensional coordinates, ξ_1, ξ_2 can be decomposed into two tangential and normal components as

$$\mathbf{u} = u^j \mathbf{t}_j,$$

where the covariant base vectors $\mathbf{t}_1, \mathbf{t}_2$ are tangent in direction of coordinate lines ξ_1, ξ_2, respectively, and $\hat{\mathbf{t}}_3$ is a unit normal vector to the undeformed mid-surface. Coefficients $u^j; j = 1, 2, 3$ are associated components of a displacement $\mathbf{u}$ in two tangential and one normal directions.

The linear version of the undeformed covariant metric tensor of the mid-surface is expressed as

$$a_{\alpha\beta} = \mathbf{t}_\alpha \cdot \mathbf{t}_\beta,$$

and, the linear version of the deformed metric tensor can be considered as

$$A_{\alpha\beta} = a_{\alpha\beta} + u^\beta|_\alpha + u^\alpha|_\beta,$$

where the quantity $u^j|_\alpha = (u^j_{,\alpha} + u^i \Gamma^j_{i\alpha})$ is the j-component of α-derivative of a displacement $\mathbf{u}$ in the tangential and normal directions. Furthermore, $\Gamma^j_{i\alpha} = \left(\frac{\partial \mathbf{t}_i}{\partial \xi_\alpha} \cdot \mathbf{t}_j\right)$ denotes Christoffel symbol of the second kind. Note that the comma preceding the subscript j signifies partial differentiation with respect to the coordinate line ξ_j. Also, a Latin index represents any of the numbers $1, 2, 3$ and a Greek index represents the numbers $1, 2$.

The determinants of the covariant metric tensor in undeformed and deformed shell are a and A, respectively, and can be calculated by the determinant of the associated metric tensors.

Then, the linearised strain tensor is expressed as

$$\gamma_{\alpha\beta} = \frac{1}{2}\left(A_{\alpha\beta} - a_{\alpha\beta}\right) = \frac{1}{2}\left(u^\beta|_\alpha + u^\alpha|_\beta\right).$$

Next, the linear version of the curvature tensor in the undeformed and deformed configurations can be specified as, respectively,

$$b_{\alpha\beta} = \hat{\mathbf{t}}_3 \cdot \mathbf{r}_{,\alpha\beta},$$

and,

$$B_{\alpha\beta} = b_{\alpha\beta} + \left[\frac{1}{a} L_j \Gamma^j_{\alpha\beta} - \frac{L_3}{a^2}\Gamma^3_{\alpha\beta}\right] + \left(u^i|_\alpha \Gamma^3_{i\beta} + u^3_{,\alpha\beta} + u^k \frac{\partial \Gamma^3_{k\alpha}}{\partial \xi_\beta} + \frac{\partial u^k}{\partial \xi_\beta}\Gamma^3_{k\alpha}\right).$$

Then, the linearised bending tensor can be obtained as

$$\kappa_{\alpha\beta} = b_{\alpha\beta} - B_{\alpha\beta} = -\left[\frac{1}{a} L_j \Gamma^j_{\alpha\beta} - \frac{L_3}{a^2}\Gamma^3_{\alpha\beta}\right] - \left(u^i|_\alpha \Gamma^3_{i\beta} + u^3_{,\alpha\beta} + u^k \frac{\partial \Gamma^3_{k\alpha}}{\partial \xi_\beta} + \frac{\partial u^k}{\partial \xi_\beta}\Gamma^3_{k\alpha}\right),$$

where L_j is defined as

$$L = \left(a_{12}u^3|_2 - a_{22}u^3|_1, -a_{11}u^3|_2 + a_{21}u^3|_1, a_{11}u^2|_2 - a_{12}u^1|_2 + a_{22}u^1|_1 - a_{21}u^2|_1\right)^T.$$

1.1.2 A finite element representation of the displacements

When we substitute all linearised terms of the strain tensor, $\gamma_{\alpha\beta}$ and the bending tensor, $\kappa_{\alpha\beta}$ and their variations back into the shell governing equation (1), it can be seen that the governing equation contains a second-order derivative of a normal displacement and a first-order derivative of tangential displacements in both directions. Therefore, we have that the normal displacement requires C^1-continuity while the tangential displacements in both directions require only C^0-continuity.

To approximate the normal displacement, a C^1-continuous interpolation function has to be considered in order to ensure the continuity of its derivatives in the finite element method. In `oomph-lib`, the Bell shape functions can be employed to provide the C^1-continuity when the straight boundary domain is concerned. Alternatively, the C^1-curved triangular shape functions can be used when the curvilinear boundary is concerned. The Bell and the C^1-curved triangular shape functions can be overloaded from `BellElementShape<>` and `C1CurvedElementShape<>`, respectively.

Unlike the normal displacement, an interpolation function approximating the solution of the tangential displacements does not require a continuity for its derivatives. Therefore, a Lagrange-type interpolation function will be employed to approximate the tangential displacements, u_1, u_2. These interpolation functions will be overloaded from `TElementShape<>`.

`Oomph-lib's BellShellElement` and `oomph-lib's C1CurvedShellElement` provide a discretisation of the variational principle (1) with two-dimensional, subparametric, triangular finite elements on a straight-sided and curvilinear boundaries, respectively. In these elements, the displacements are regarded as unknowns, and the C^1-interpolation is used to interpolate the normal displacement while the Lagrange polynomials is used to interpolate the tangential displacements. Furthermore, the geometry is approximated by the linear Lagrange interpolations for a straight-sided boundary domain while the cubic polynomial is employed to approximate the curved boundary domain.

1.2 Numerical results

In this section, implementations of the linearised governing equations for thin-shell problems will be illustrated. There are three problems considered in this document; square- and circular-plate, and circular tube bending. The implementations will be based on the governing equations (1) that we derived in section 1.1.

In all cases, the problems will be solved with the assumption that the thickness is thin so that the linear theory can be applied. Our choice of thickness is 0.01. Also, applied forces will be applied in normal direction to the undeformed surface with no initial stress.

Furthermore, in order to perform the finite element implementations, the domains of interest will be discretised by triangular elements with an unstructured mesh. In this study, `Oomph-lib's BellShellElement` is employed when straight-sided boundaries are concerned while `Oomph-lib's C1CurvedShellElement` is employed when curvilinear boundaries are involved.

Note that figures of the displacements that we will illustrate throughout this document will be in the Cartesian coordinate system. Since the displacements obtained as the solution of (1) are in the tangential and normal coordinates, the transformation to the Cartesian coordinate system has to be done by

$$u_x = u^1 t_1^1 + u^2 t_2^1 + u^3 \hat{t}_3^1, \quad u_y = u^1 t_1^2 + u^2 t_2^2 + u^3 \hat{t}_3^2, \quad u_z = u^1 t_1^3 + u^2 t_2^3 + u^3 \hat{t}_3^3,$$

where $t_i^j; j = 1, 2, 3$, denote components of the tangent base vector $\mathbf{t}_i; i = 1, 2$, and the unit normal vector, $\hat{\mathbf{t}}_3$, respectively.

1.2.1 The square-plate bending problem

Here, we will consider a deformation of a flat plate which is subjected to a pressure loading on its upper surface as illustrated in the following figure. The length of the plate in both directions is 1. The boundary conditions in this problem are two clamped boundaries, $\xi_1 = 0, \xi_1 = 1$, and two free edges, $\xi_2 = 0, \xi_2 = 1$. Therefore, the displacement and its rotational degrees of freedom are pinned at the boundaries $\xi_1 = 0, \xi_1 = 1$ while all degrees of freedom are set to be free at $\xi_2 = 0, \xi_2 = 1$.

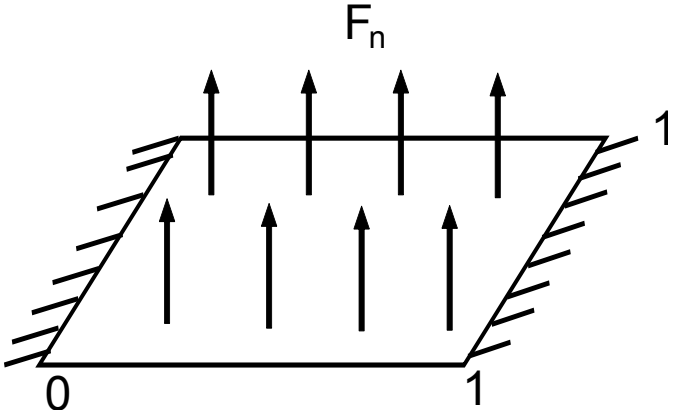

Figure 1.2 The geometry of the square plate with two clamped edges and two free edges. Forces applied to a body are uniform in the outward normal direction.

In order to implement the finite element method of a two-dimensional space in this study, the domains of interest which is the unit square will be discretised by triangles. Note that the unstructured mesh contains 150 elements.

Here are figures illustrate displacements in all directions in Cartesian coordinates system for the flat plate problem stated above with the applied loads in the normal direction equal to 1.0×10^{-8}.

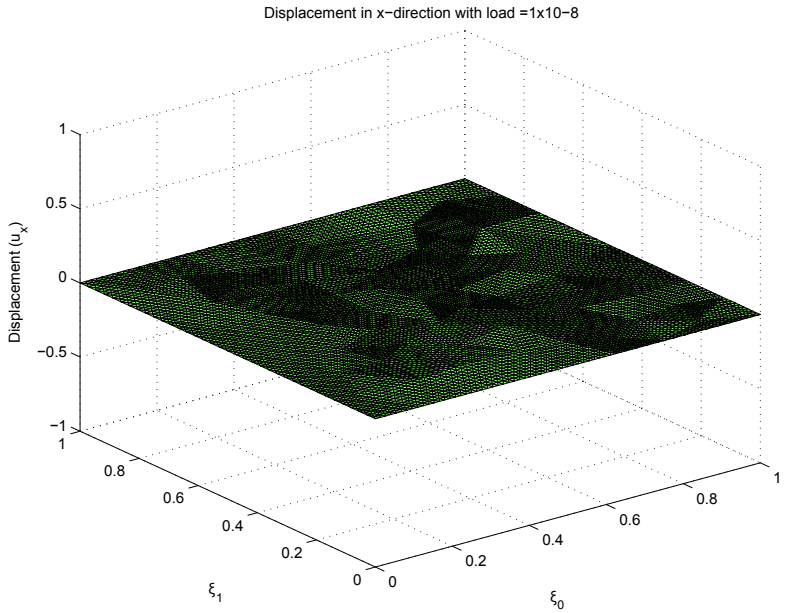

Figure 1.3 The displacement in x-direction.

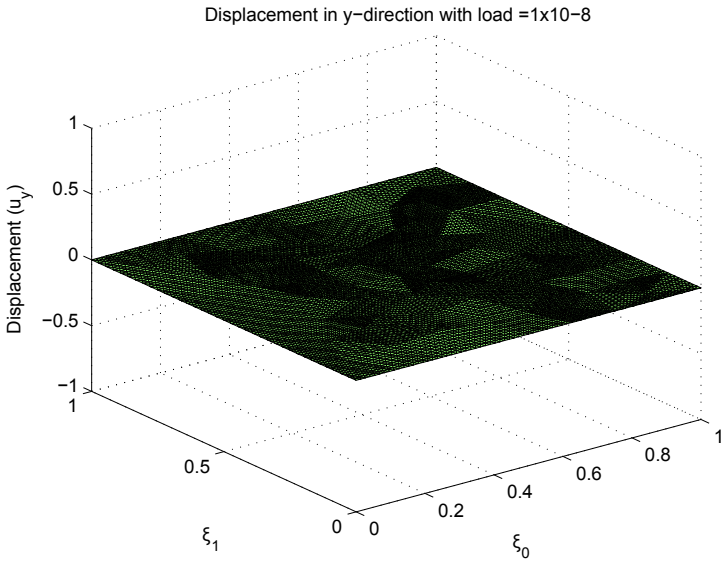

Figure 1.4 The displacement in y-direction.

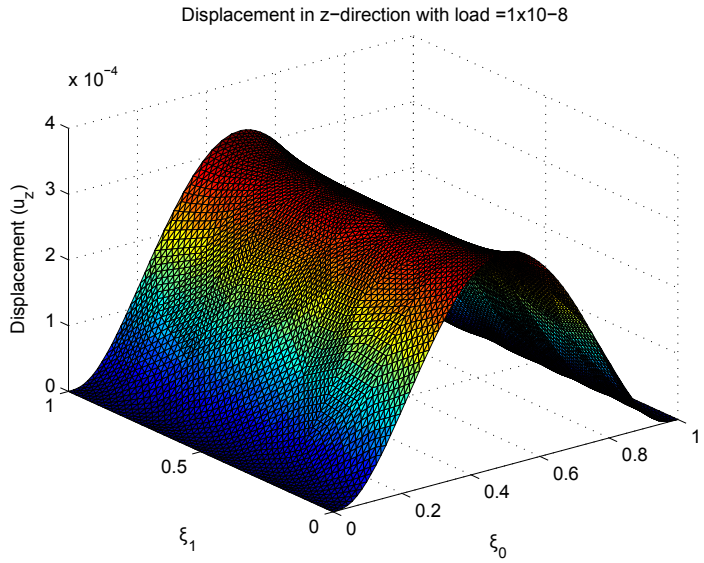

Figure 1.5 The displacement in z-direction.

Regarding the displacements in both tangential directions, it can be seen from Figures 1.3 and 1.4 that no deformation occurs in the x- and y-directions. The underlying reason is that the forces are applied in the normal direction to the surface of the plate which correspond to the z-direction. Hence, there is no force applied in both tangential directions which correspond to the x- and y-directions. Therefore, there is no contribution to make the body deforms in those directions as the linear governing equations are not coupled between displacements in each direction. To see this, the reader have to expand (1) after substituting $\gamma_{\alpha\beta}$, $\kappa_{\alpha\beta}$, and their variations in.

1.2.2 The circular-tube problem

In this section, we consider a deformation of a circular tube which is subjected to a pressure loading on its surface as illustrated in the following figure. The loads applied on the tube are uniformly distributed in the normal direction.

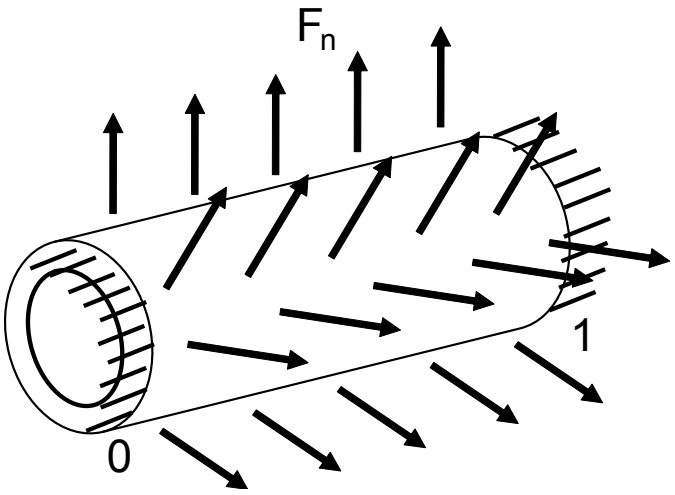

Figure 1.6 The geometry of the square plate with two clamped edges and two free edges.

To implement the deformation of the circular tube in this study, the quarter of a circular tube will be implemented and symmetric conditions are assumed along the tube. The boundary conditions determined in this problem are clamped supports at both ends of the tube.

Similar to the plate bending problem, the domains of interest will be discretised by an unstructured triangular mesh with 248 elements.

Here are figures illustrate displacements in all directions in Cartesian coordinates system for the circular tube problem stated above with the applied loads in the normal direction equal to 1.0×10^{-6}.

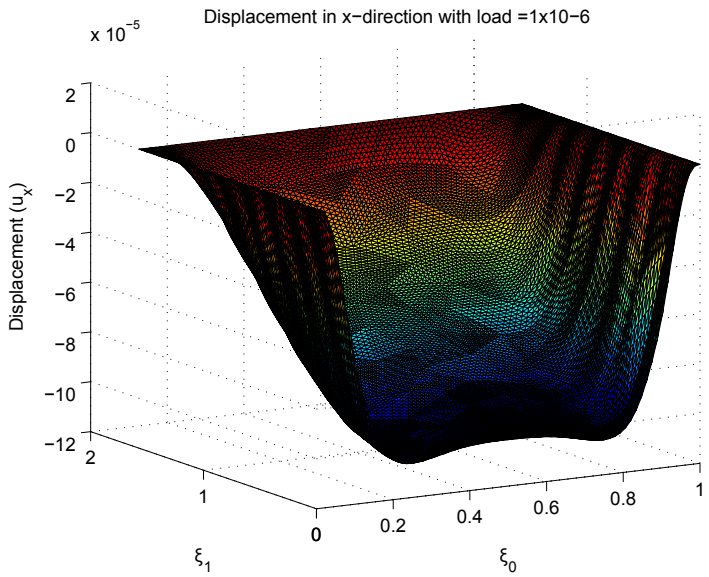

Figure 1.7 The displacement in x-direction.

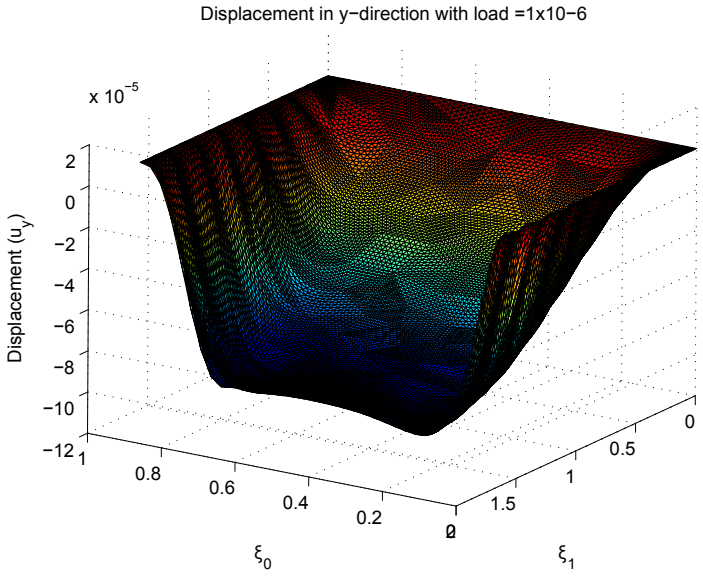

Figure 1.8 The displacement in y-direction.

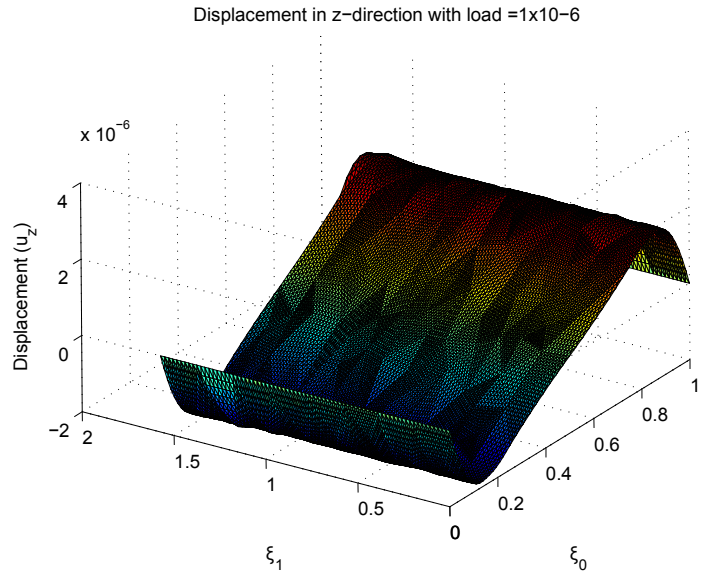

Figure 1.9 The displacement in z-direction.

It can be seen that the displacement in x- and y-directions are symmetry. This behaviour depicts that the thin-circular tube deforms axisymmetrically within a small-strain regime.

1.2.3 The circular-plate bending problem

Here, we will consider a deformation of a flat plate which is subjected to a pressure loading on its upper surface like in section 1.2.1. However, the domain of interest considered here will be curved. Therefore, the unit circular plate is considered with clamped boundaries. Note that only one quarter of the unit circular plate is analysed and symmetric conditions are applied in this problem.

In order to implement the finite element method of a two-dimensional space in this study, the domains of interest which is the unit circular plate will be discretised by triangles. Note that the unstructured mesh contains 84 elements.

Here are figures illustrate displacements in all directions in Cartesian coordinates system for the flat plate problem stated above with the applied loads in the normal direction equal to 1.0×10^{-7}.

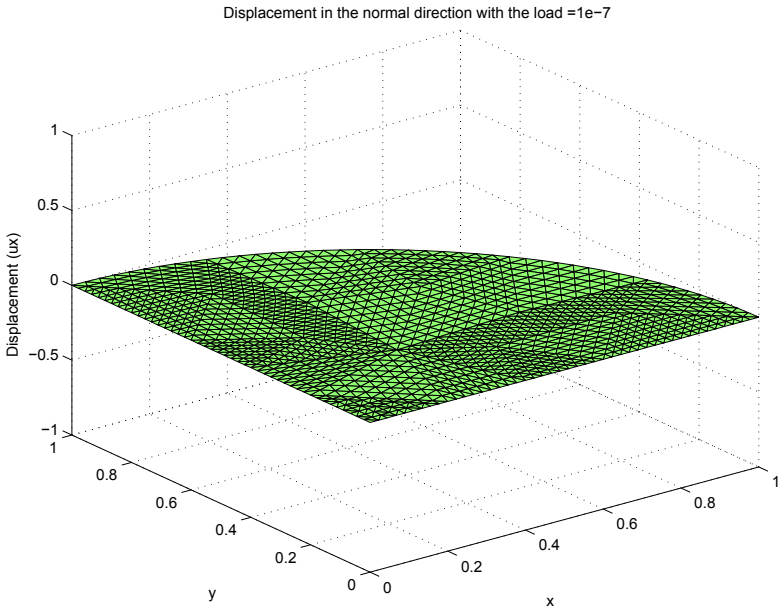

Figure 1.10 The displacement in x-direction.

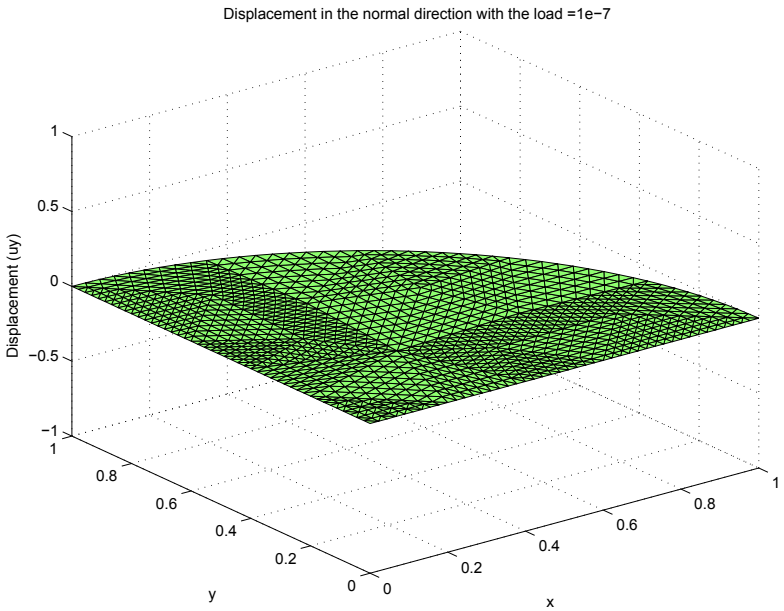

Figure 1.11 The displacement in y-direction.

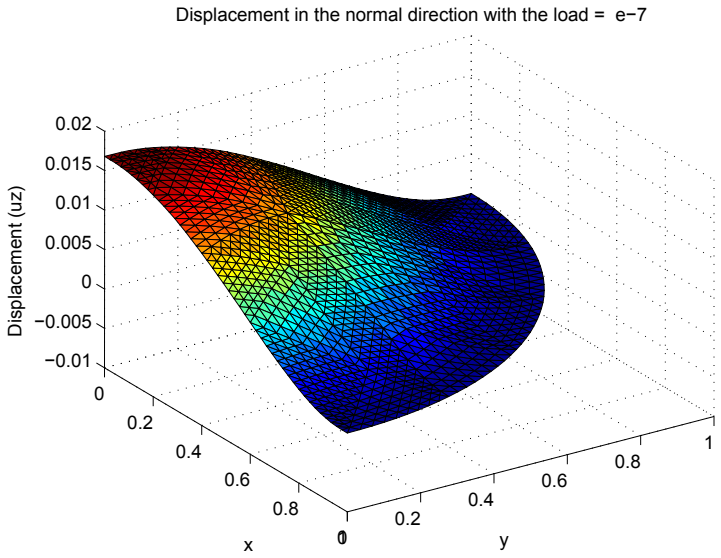

Figure 1.12 The displacement in z-direction.

Similarly, there is no deformation in both x- and y-directions as explained in section 1.2.1.

1.3 Implementation in oomph-lib

In the following, we illustrate the driver codes for the square-plate bending problem while other problems can be determined similarly.

1.3.1 Global parameters and functions

The namespace `Physical_Variables` is where the source function and the exact solution are defined. The source function can be specified via `Physical_Variables::source_function()` while the exact solutions are defined via `Physical_Variables::get_exact_u()`. Note that the six exact solutions correspond to the six degrees of freedom defined on each node. Furthermore, the applied source functions are required to be in the direction of the unit normal vector.

```
//======================start_of_namespace============================
/// Namespace for the solution of 2D linear shell equation
//====================================================================
namespace Physical_Variables
{
 /// Pressure load
 double P_ext;

 double epsilon = 1.0e-8;

 /// Exact solution as a vector
 /// differentiate u with respect to global coordinates
 void get_exact_u(const Vector<double>& x, Vector<double>& u)
 {
  u[0] = 0.0;
  u[1] = 0.0;
  u[2] = 0.0;
  u[3] = 0.0;
  u[4] = 0.0;
  u[5] = 0.0;
  u[6] = 0.0;
  u[7] = 0.0;
 }
```

```
 /// Source function applied in the normal vector
 void source_function(const Vector<double>& x, const Vector<double>& unit_n, Vector<double>&
       source)
 {
  for(unsigned i=0;i<3;i++)
   {
    source[i] = epsilon*P_ext*unit_n[i];
   }
 }
} // end of namespace
```

1.3.2 The driver code

The driver code is very simple and short. It is where the problem is defined. In this study, the problem is constructed using the unstructured mesh with triangular elements in 2D. A number of nodes in an element has to be specific as a template parameter in the problem set up. This is crucial in order to take care of element nodes when the number of nodes on the element defined differently for each interpolation functions.

Normally, in the problem that C^1-shape functions are the only interpolation functions used to approximate the variable space, the number of `NNODE_1D` remains 2 as required by the C^1-shape functions (see the Bell triangular finite element and the C^1-curved triangular finite element tutorials). However, in a linear shell problem, different shape functions are used to interpolate displacements in different directions as different order of continuity is required in their governing equations.

In this study, the displacements in tangential directions are chosen to approximate by the quadratic Lagrange interpolations which provide only C^0 continuity and are defined to have 3 nodes per side on a triangle. On the other hand, the approximation of the normal displacement employs the C^1-interpolation functions which are defined to have 2 nodes per side on a triangle.

Therefore, the number of `NNODE_1D` has been modified in the `BellShellElement<DIM,NNODE_1D>` (and `C1CurvedShellElement` when dealing with the curvilinear boundary domain) to be 3 in this problem. Consequencely, the extra nodes that are not necessary for the C^1-shape functions have to be taken care.

Following the usual self-test, we call the function `MyLinearisedShellProblem::parameter_study()` to compute the solution of the problem within a range of external pressures.

```
//======start_of_main=====================================================
/// Driver for 2D linearised shell problem: square flat plate
//========================================================================
int main(int argc, char* argv[])
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);

 // Check number of command line arguments: Need exactly two.
 if (argc!=4)
  {
   std::string error_message =
    "Wrong number of command line arguments.\n";
   error_message +=
    "Must specify the following file names  \n";
   error_message +=
    "filename.node then filename.ele then filename.poly\n";

   throw OomphLibError(error_message,
                       "main()",
                       OOMPH_EXCEPTION_LOCATION);
  }
 // Convert arguments to strings that specify th input file names
 string node_file_name(argv[1]);
 string element_file_name(argv[2]);
 string poly_file_name(argv[3]);

 // Set up the problem:
 MyLinearisedShellProblem<BellShellElement<2,3>,2,3> //
      Element type as template parameter
  problem(Physical_Variables::source_function,node_file_name,
      element_file_name,poly_file_name);
```

```
 // Check whether the problem can be solved
 cout << "\n\n\nProblem self-test ";
 if (problem.self_test()==0)
  {
   cout << "passed: Problem can be solved." << std::endl;
  }
 else
  {
   throw OomphLibError("failed!","main()",OOMPH_EXCEPTION_LOCATION);
  }

 // Solve the problem
 problem.parameter_study();
} //end of main
```

1.3.3 The problem class

The problem class has five member functions, illustrated as follows:

- The problem constructor
- `action_before_newton_solve()` : Update the problem specifications before solve. Boundary conditions maybe set here.
- `action_after_newton_solve()` : Update the problem specifications after solve.
- `doc_solution()` : Pass the number of the case considered, so that output files can be distinguished.
- `parameter_study()` : Computes the shell's deformation for a range of external pressures.

From the above mentioned functions, only the problem constructor is non-trivial. The reader is referred to `another tutorial` for a description on `parameter_study`.

In the present problem, the function `Problem::actions_after_newton_solve()` is not required, so it remains empty. Also, the class includes two private data members which store pointers to a source function and to the geometric object that specifies the shell's undeformed shape.

```
//==start_of_problem_class============================================
/// 2D linearised shell problem.
//====================================================================
template<class ELEMENT, unsigned DIM, unsigned NNODE_1D>
class MyLinearisedShellProblem : public Problem
{

public:

 /// Constructor: Pass number of elements and pointer to source function
 MyLinearisedShellProblem(typename MyShellEquations<DIM,NNODE_1D>::SourceFctPt
      source_fct_pt,
                          const string& node_file_name,
                          const string& element_file_name,
                          const string& poly_file_name);

 /// Destructor (empty)
 ~MyLinearisedShellProblem()
  {
   delete mesh_pt();
  }

 /// Update the problem specs before solve: (Re)set boundary conditions
 void actions_before_newton_solve();

 /// Update the problem specs after solve (empty)
 void actions_after_newton_solve(){}

 /// \short Doc the solution, pass the number of the case considered,
 /// so that output files can be distinguished.
 void doc_solution(DocInfo& doc_info);
```

```
 void parameter_study();

private:

 /// Pointer to source function
 typename MyShellEquations<DIM,NNODE_1D>::SourceFctPt Source_fct_pt;
 /// Pointer to geometric object that represents the shell's undeformed shape
 GeomObject* Undef_midplane_pt;

}; // end of problem class
```

1.3.4 The Problem constructor

The problem constructor starts by overloading the function `Problem::mesh_pt()` and set to the specific mesh used in the problem. In this tutorial, we implement the problem with 2D triangular unstructured mesh which is externally created by `Triangle`. The generated output will be used to build `oomph-lib` mesh. The reader may refer to another tutorial to create an unstructured triangular mesh internally.

```
//=====start_of_constructor=============================================
/// \short Constructor for 2D Shell problem.
/// Discretise the 2D domain with n_element elements of type ELEMENT.
/// Specify function pointer to source function.
//======================================================================
template<class ELEMENT, unsigned DIM, unsigned NNODE_1D>
MyLinearisedShellProblem<ELEMENT,DIM,NNODE_1D>::MyLinearisedShellProblem
(typename MyShellEquations<DIM,NNODE_1D>::SourceFctPt source_fct_pt,
const string& node_file_name,const string& element_file_name,
const string& poly_file_name) :
 Source_fct_pt(source_fct_pt)
{
 // Build mesh and store pointer in Problem
 Problem::mesh_pt() = new TriangleMesh<ELEMENT>(node_file_name,element_file_name,poly_file_name);
```

We then create the undeformed centreline of the shell to one of an oomph-lib's standard shell geometric objects.

```
 // set the undeformed shell
 Undef_midplane_pt = new Plate(1.0,1.0);
```

Prior to consider the boundary conditions, we will illustrate how to take care extra nodes on the element for the C^1-interpolations. Since there is no degree of freedom of the C^1-interpolations defined on the mid-sided nodes, they have to be pinned.

In order to do this, firstly, we loop over the element and pin all degrees of freedom on non-vertex nodes that associated with the C^1-interpolations. Since nodes in the element situate anticlockwise and the midside nodes fill in progressing along the consecutive edges, the pinning procedure is easily done by starting to pin from the third node and so on.

```
 // pinning the middle nodes in each element for normal direction
 for(unsigned n=0;n<n_element;n++)
 {
  // Upcast from GeneralisedElement to the present element
  ELEMENT *elem_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(n));

  unsigned nnode = elem_pt->nnode();
  for(unsigned i=0;i<nnode;i++)
   {
    if((i==0) || (i==1) || (i==2))
     {
     }
    else
     {
      elem_pt->node_pt(i)->pin(2);
      elem_pt->node_pt(i)->pin(3);
      elem_pt->node_pt(i)->pin(4);
      elem_pt->node_pt(i)->pin(5);
      elem_pt->node_pt(i)->pin(6);
      elem_pt->node_pt(i)->pin(7);
     }
   }
 } // end of the middle node pinning
```

Next, the boundary conditions of the problem will be taken care. We pin the nodal values on the boundaries where the boundary conditions applied. Note that, at the clamped boundaries, all second-order derivatives degrees of freedom are also pinned in order to reduce the number of degrees of freedom in the problem. These second-order derivatives are the derived boundary conditions that can be taken care by the natural boundary conditions.

```
 // start_of_boundary_conditions
 // Set the boundary conditions for this problem: By default, all nodal
 // values are free -- we only need to pin the ones that have
 // Dirichlet conditions.
// unsigned n_side0 = mesh_pt()->nboundary_node(0);
 unsigned n_side1 = mesh_pt()->nboundary_node(1);
 //unsigned n_side2 = mesh_pt()->nboundary_node(2);
 unsigned n_side3 = mesh_pt()->nboundary_node(3);

 //------ PLATE BENDING PROBLEM -------------
 // Pin the single nodal value at the single node on mesh
 // boundary 1 (= the right domain boundary at x=1)
 /// loop over the nodes on the boundary
 for(unsigned i=0;i<n_side1;i++)
  {
   // loop over the degrees of freedom that need to be
   // taken care for the Dirichlet boundary conditions
   for(unsigned j=0;j<8;j++)
    {
     mesh_pt()->boundary_node_pt(1,i)->pin(j);
    }
  }

 // boundary 3 (= the left domain boundary at x=0)
 /// loop over the nodes on the boundary
 for(unsigned i=0;i<n_side3;i++)
  {
   // loop over the degrees of freedom that need to be
   // taken care for the Dirichlet boundary conditions
   for(unsigned j=0;j<8;j++)
    {
     mesh_pt()->boundary_node_pt(3,i)->pin(j);
    }
  } // end of boundary conditions
```

We then loop over the elements and set the pointer to the physical parameters (if any), the function pointer to the source function, and the pointer to the geometric object that specifies the undeformed surface of the shell.

```
 // Loop over elements and set pointers to Physical parameters
 for(unsigned i=0;i<n_element;i++)
  {
   // Upcast from GeneralisedElement to the present element
   ELEMENT *elem_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));

   //Set the source function pointer and all physical variables
   elem_pt->source_fct_pt() = Source_fct_pt;

   //Set the pointer to the undeformed mid-plane geometry
   elem_pt->undeformed_midplane_pt() = Undef_midplane_pt;
  } // end of loop over elements
```

We finish the constructor by assigning the equation numbering scheme.

```
 // Setup equation numbering scheme
 assign_eqn_numbers();

} // end of constructor
```

1.3.5 Action before newton solve

In the `action_before_newton_solve()`, the problem specifications will be updated before performing the newton solve. The boundary values will be (re)set from the exact solutions.

```
//===start_of_actions_before_newton_solve=================================
/// \short Update the problem specs before solve: (Re)set boundary values
/// from the exact solution.
//========================================================================
template<class ELEMENT, unsigned DIM, unsigned NNODE_1D>
void MyLinearisedShellProblem<ELEMENT,DIM,NNODE_1D>::actions_before_newton_solve
      ()
{
 // Assign boundary values for this problem by reading them out
 // from the exact solution.
 for(unsigned n=0;n<mesh_pt()->nboundary();n++)
  {
   // find number of nodes in each boundary
   unsigned n_side = mesh_pt()->nboundary_node(n);
   /// loop over the nodes on the boundary
   for(unsigned j=0;j<n_side;j++)
    {
     // Left boundary is every nodes on the left boundary
     Node* left_node_pt=mesh_pt()->boundary_node_pt(n,j);

     // Loop for variables u
     ELEMENT e;
     Vector<double> u((e.required_nvalue(0)));
     for(unsigned i=0;i<(e.required_nvalue(0));i++)
      {
       // Determine the position of the boundary node (the exact solution
       // requires the coordinate in a 1D vector!)
       Vector<double> x(2);
       x[0]=left_node_pt->x(0);
       x[1]=left_node_pt->x(1);

       // Boundary value (read in from exact solution)
       Physical_Variables::get_exact_u(x,u);

       // Assign the boundary condition to nodal values
       left_node_pt->set_value(i,u[i]);
      }
    }
  }
} // end of actions before solve
```

1.4 Source files for this tutorial

- The source files for this tutorial are located in the directory:

  demo_drivers/shell/

- The driver code for the square plate bending problem is:

  demo_drivers/shell/plate/unstructured_clamped_square_plate.cc

- The driver code for the circular tube problem is:

  demo_drivers/shell/clamped_shell/unstructured_clamped_curved_shell.cc

- The driver code for the circular plate bending problem is:

demo_drivers/shell/circular_plate/unstructured_clamped_circular_plate.↩
cc

1.5 PDF file

A pdf version of this document is available.