Chapter 1

Demo problem: Deformation of a string under tension, using Kirchhoff-Love Beam elements.

In this document, we discuss the solution of a simple one-dimensional solid mechanics problem, using `oomph-lib's` Kirchhoff-Love beam elements:

One-dimensional beam problem

Determine the deformation of a pre-stressed, pressure-loaded elastic beam of length L^* and wall thickness h^* . In its undeformed configuration, the beam is subject to an axial (2nd Piola-Kirchhoff) pre-stress of size σ_0^* . We wish to compute its deformation when it is loaded by a spatially uniform pressure of size p_{ext}^* .

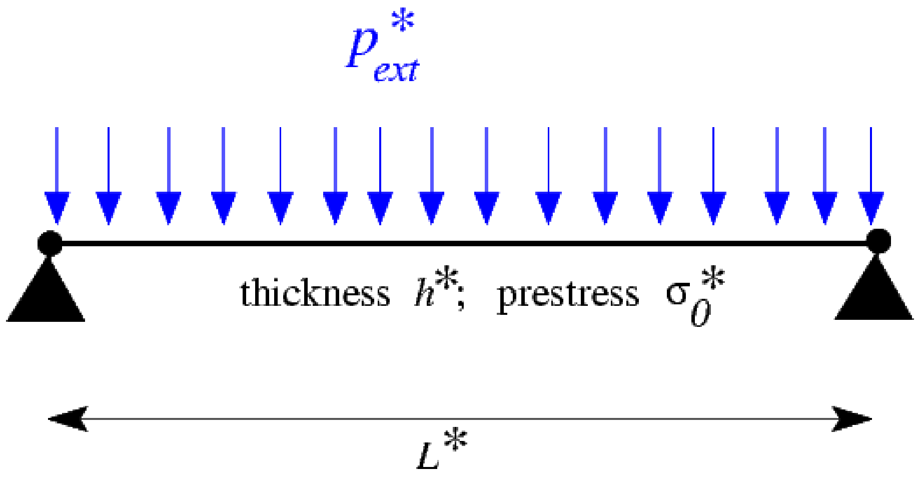

Figure 1.1 A pre-stressed elastic beam under constant pressure loading.

1.1 Theory and non-dimensionalisation

`oomph-lib's` beam elements are based on geometrically nonlinear Kirchhoff-Love beam theory with incrementally linear constitutive equations (Young's modulus E and Poisson's ratio ν). The equations are implemented in non-dimensional form, obtained by non-dimensionalising all length on some lengthscale $\mathcal{L}$, by scaling the stresses and the applied traction on the beam's effective Young's modulus $E_{eff} = E/(1-\nu^2)$ and by non-dimensionalising time on some timescale $\mathcal{T}$ so that the dimensional (identified by an asterisk) and non-dimensional variables are related by

$$L^* = \mathcal{L}\, L,$$

$$h^* = \mathcal{L}\, h,$$

$$t^* = \mathcal{T}\, t,$$

$$p^*_{ext} = E_{eff}\, p_{ext},$$

and

$$\sigma_0^* = E_{eff}\, \sigma_0.$$

The beam's undeformed shape is parametrised by a non-dimensional Lagrangian coordinate $\xi = \xi^*/\mathcal{L}$ so that the non-dimensional position vector to a material particle on the beam's centreline in the undeformed configuration is given by $\mathbf{r}_w(\xi)$. We denote the unit normal to the beam's undeformed centreline by $\mathbf{n}$. The applied traction $\mathbf{f} = \mathbf{f}^*/E_{eff}$ (a force per unit deformed length of the beam) deforms the beam, causing its material particles to be displaced to their new positions $\mathbf{R}_w(\xi)$; the unit normal to the beam's deformed centreline is $\mathbf{N}$.
The non-dimensional form of the principle of virtual displacements that governs the beams deformation is then given by

$$\int_0^L \left[(\sigma_0 + \gamma)\, \delta\gamma + \frac{1}{12} h^2 \kappa\, \delta\kappa - \left(\frac{1}{h}\sqrt{\frac{A}{a}}\, \mathbf{f} - \Lambda^2 \frac{\partial^2 \mathbf{R}_w}{\partial t^2} \right) \cdot \delta\mathbf{R}_w \right] \sqrt{a}\, d\xi = 0, \qquad (1)$$

where

$$a = \frac{\partial \mathbf{r}_w}{\partial \xi} \cdot \frac{\partial \mathbf{r}_w}{\partial \xi},$$

and

$$A = \frac{\partial \mathbf{R}_w}{\partial \xi} \cdot \frac{\partial \mathbf{R}_w}{\partial \xi},$$

represent the squares of the lengths of infinitesimal material line elements in the undeformed and the deformed configurations, respectively.

$$ds = \sqrt{a}\, d\xi \quad \text{and} \quad dS = \sqrt{A}\, d\xi.$$

A and a may be interpreted as the "1x1 metric tensors" of the beam's centreline, in the deformed and undeformed configurations, respectively. The ratio $\sqrt{A/a}$ represents the "extension ratio" or the "stretch" of the beam's centreline.
We represent the curvature of the beam's centreline before and after the deformation by

$$b = \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}_w}{d\xi^2}$$

and

$$B = \mathbf{N} \cdot \frac{\partial^2 \mathbf{R}_w}{d\xi^2}$$

respectively. The ("1x1") strain and and bending "tensors" γ and κ are then given by

$$\gamma = \frac{1}{2}(A - a) \quad \text{and} \quad \kappa = -(B - b).$$

Finally,

$$\Lambda = \frac{\mathcal{L}}{\mathcal{T}} \sqrt{\frac{\rho}{E_{eff}}}$$

is the ratio of the natural timescale of the beam's in-plane extensional oscillations,

$$\mathcal{T}_{natural} = \mathcal{L} \sqrt{\frac{\rho}{E_{eff}}},$$

to the timescale $\mathcal{T}$ used in the non-dimensionalisation of the equations. Λ^2 may be interpreted as the non-dimensional wall density, therefore $\Lambda = 0$ corresponds to the case without wall inertia.

`oomph-lib's HermiteBeamElement` provides a discretisation of the variational principle (1) with one-dimensional, isoparametric, two-node Hermite solid mechanics elements. In these elements, the Eulerian positions of the nodes, accessible via `Node::x(...)`, are regarded as unknowns, and Hermite interpolation is used to interpolate the position between the nodes, so that the Eulerian position of material points within an element is given by

$$x_i(s) = \sum_{j=1}^{2}\sum_{k=1}^{2} X_{ijk}\ \psi_{jk}(s) \quad \text{for } i = 1, 2, \qquad (2)$$

where $s \in [-1, 1]$ is the element's 1D local coordinate. The functions $\psi_{jk}(s)$ are the one-dimensional Hermite shape functions

$$\psi_{11}(s) = \frac{1}{4}\left(s^3 - 3s + 2\right),$$

$$\psi_{12}(s) = \frac{1}{4}\left(s^3 - s^2 - s + 1\right),$$

$$\psi_{21}(s) = \frac{1}{4}\left(2 + 3s - s^3\right),$$

$$\psi_{22}(s) = \frac{1}{4}\left(s^3 + s^2 - s - 1\right).$$

They have the property that

- $\psi_{j1} = 1$ at node j and $\psi_{j1} = 0$ at the other node. Furthermore, $d\psi_{j1}/ds = 0$ at both nodes.
- $d\psi_{j2}/ds = 1$ at node j and $d\psi_{j2}/ds = 0$ at the other node. Furthermore, $\psi_{j2} = 0$ at both nodes.

The mapping (2) therefore provide an independent interpolation for the position and its derivative with respect to the local coordinate s. As a result we have two types of (generalised) nodal coordinates:

- X_{ij1} represents the i-th coordinate of the element's local node j.
- X_{ij2} represents the derivative of i-th coordinate with respect to the local coordinate s, evaluated at the element's local node j.

This representation ensures the C^1 continuity of the wall shape, required by the variational principle (1) which contains second derivatives of $\mathbf{R}_w$. Physically, the inter-element continuity of the slope is enforced by the beam's nonzero bending stiffness.

The two "types" of "generalised" positional degrees of freedom are accessible via the function `Node::x_↩ gen(k,i)`, where `Node::x_gen(0,i)` $\equiv$ `Node::x(i)`. The two types of positional degrees of freedom correspond to the two types of boundary conditions (pinned and clamped) that can be applied at the ends of the beam. Mathematically, the number of boundary conditions reflects the fact that the Euler-Lagrange equations of the variational principle are of fourth-order.

The nodes of solid mechanics elements are `SolidNodes`, a generalisation of the basic `Node` class that allows the nodal positions to be treated as unknowns. Its member function `SolidNode::pin_position(...)` allows the application of boundary conditions for the (generalised) nodal positions. The table below lists several common boundary conditions and their application:

Boundary condition	Mathematical condition	Implementation

Pinned:	$\mathbf{R}_w \cdot \mathbf{e}_x = const.$
	$\mathbf{R}_w \cdot \mathbf{e}_y = const.$
		`SolidNode::pin_position(0);`
		`SolidNode::pin_position(1);`
		
		or, equivalently,
		
		`SolidNode::pin_position(0,0);`
		`SolidNode::pin_position(0,1);`

Pinned, sliding in the x-direction:	$\mathbf{R}_w \cdot \mathbf{e}_y = const.$	`SolidNode::pin_position(1);`
		
		or, equivalently,
		
		`SolidNode::pin_position(0,1);`

Pinned, sliding in the y-direction:	$\mathbf{R}_w \cdot \mathbf{e}_x = const.$	`SolidNode::pin_position(0);`
		
		or, equivalently,
		
		`SolidNode::pin_position(0,0);`

Clamped:	$\mathbf{R}_w \cdot \mathbf{e}_x = const.$
	$\mathbf{R}_w \cdot \mathbf{e}_y = const.$
	$d(\mathbf{R}_w \cdot \mathbf{e}_y)/d\xi = 0.$
		`SolidNode::pin_position(0);`
		`SolidNode::pin_position(1);`
		`SolidNode::pin_position(1,1);`
		
		or, equivalently,
		
		`SolidNode::pin_position(0,0);`
		`SolidNode::pin_position(0,1);`
		`SolidNode::pin_position(1,1);`

Clamped, sliding in the y-direction (symmetry boundary condition!)	$\mathbf{R}_w \cdot \mathbf{e}_x = const.$
	$d(\mathbf{R}_w \cdot \mathbf{e}_y)/d\xi = 0.$
		`SolidNode::pin_position(0);`
		`SolidNode::pin_position(1,1);`
		
		or, equivalently,
		
		`SolidNode::pin_position(0,0);`
		`SolidNode::pin_position(1,1);`

The `HermiteBeamElement` provides default values for all non-dimensional physical parameters:

- the non-dimensional 2nd Piola Kirchhoff pre-stress σ_0 is zero.
- the non-dimensional beam thickness h is 1/20.
- the timescale ratio Λ^2 is 1.
- the non-dimensional traction vector $\mathbf{f}$ evaluates to zero.

These values can be over-written via suitable access functions. [Time-dependent computations also require the specification of a timestepper for the elements. This is demonstrated in another example.] The "user" must specify:

- the undeformed wall shape $\mathbf{r}_w(\xi)$ as a `GeomObject` – see the earlier example for a discussion of `oomph-lib's` geometric objects.

1.2 Results

The animation shown below illustrates the deformation of the beam under increasing pressure. An increase in pressure initially deflects the beam vertically downwards. The pressure acts as a "follower load" since it always acts in the direction normal to the deformed beam. This causes the beam to deform into an approximately circular shape whose radius increases rapidly.

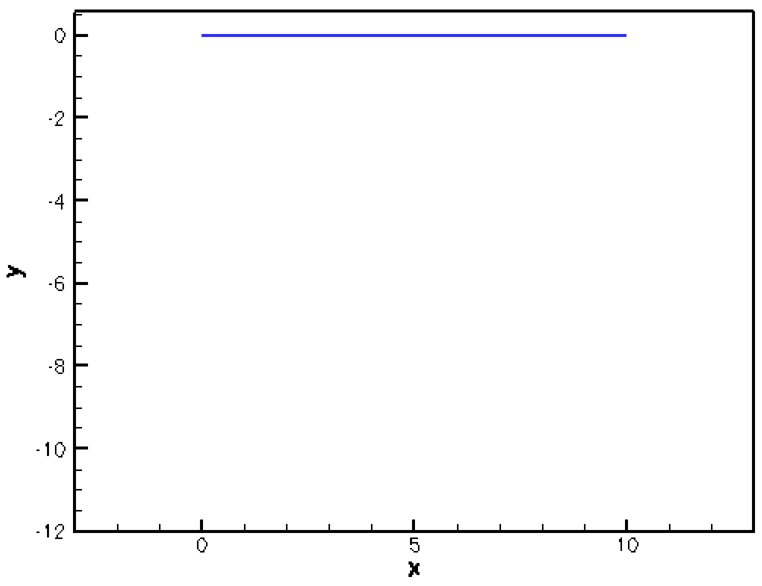

Figure 1.2 Deformation of a pre-stressed elastic beam under constant pressure loading.

1.3 An approximate analytical solution

Before discussing the details of the numerical solution, we will derive an approximate analytical solution of the problem. The analytical solution will be useful to validate the computational results, and the derivation will allow us to discuss certain aspects of the theory in more detail.
We start by parametrising the beam's undeformed shape as

$$\mathbf{r}_w(\xi) = \begin{pmatrix} \xi \\ 0 \end{pmatrix} \qquad \text{where } \xi \in [0, L].$$

The 1x1 "metric tensor" associated with this parametrisation is given by

$$a = \frac{\mathbf{r}_w(\xi)}{d\xi} \cdot \frac{\mathbf{r}_w(\xi)}{d\xi} = 1,$$

consistent with the fact that the Lagrangian coordinate ξ is the arclength along the beam's undeformed centreline.
If the beam is thin (so that $h \ll 1$) bending effects will be confined to thin boundary layers near its ends. The beam will therefore behave (approximately) like a "string under tension" and its deformed shape will be an arc of a circle. All material line elements will be stretched by the same amount so that the tension is spatially uniform. The position vector to the material particles on the beam's deformed centreline is therefore given by

$$\mathbf{R}_w(\xi) = \begin{pmatrix} \frac{1}{2}\left(1 + \frac{\sin(-\alpha/2+\alpha\xi/L)}{\sin(\alpha/2)}\right) \\ \frac{\cos(-\alpha/2+\alpha\xi/L)-\cos(\alpha/2)}{2\sin(\alpha/2)} \end{pmatrix} \qquad \text{where } \xi \in [0, L]. \qquad (3)$$

(See the Exercises and Comments for a more detailed discussion of this analytical solution.) Here α is the opening angle of the circular arc as shown in this sketch:

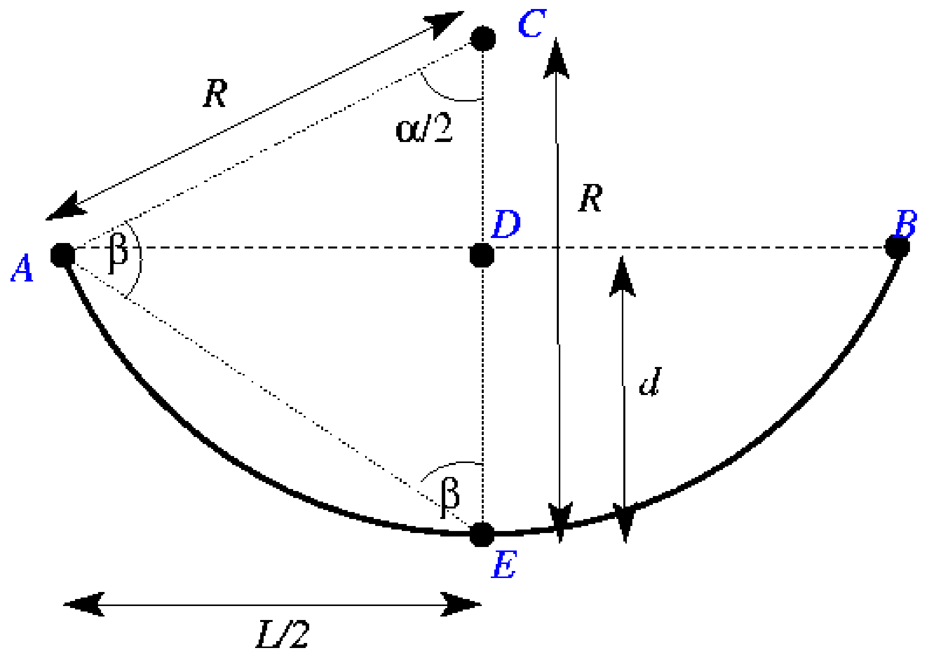

Figure 1.3 Sketch illustrating the approximate analytical solution for the problem.

This deformation generates a uniform stretch of

$$\frac{dS}{ds} = \sqrt{\frac{A}{a}} = \frac{1}{2}\frac{\alpha}{\sin(\alpha/2)},$$

corresponding to a uniform strain

$$\gamma = \frac{1}{2}\left(\frac{1}{4}\frac{\alpha^2}{\sin^2(\alpha/2)} - 1\right).$$

The incremental Hooke's law predicts a linear relation between the (2nd Piola Kirchhoff) stress, σ, and the (Green) strain, γ, so that

$$\sigma = \sigma_0 + \gamma,$$

where σ_0 is the axial pre-stress that acts in the undeformed configuration.
An elementary force balance shows that the pressure p_{ext} required to deform the beam into the shape specified by (3), is given by "Laplace's law"

$$p_{ext} = \frac{\mathtt{T}}{R}$$

where

$$R = \frac{L}{2\sin(\alpha/2)}$$

is the radius of the circular arc, and the axial tension $\mathtt{T}$ is given by

$$\mathtt{T} = h\,(\sigma_0 + \gamma)\left|\frac{d\mathbf{R}_w}{d\xi}\right| = h\,(\sigma_0 + \gamma)\,\sqrt{A}.$$

In this expression we have used the fact that the 2nd Piola Kirchhoff stress σ decomposes the (physical) stress vector into the vector $d\mathbf{R}_w/d\xi$. This vector is tangent to the deformed centreline but not necessarily a unit vector.
The pressure required to deform the beam into a circular arc with opening angle α is therefore given by

$$p_{ext} = \frac{\alpha h}{L}\left(\sigma_0 + \frac{1}{2}\left(\frac{1}{4}\frac{\alpha^2}{\sin^2(\alpha/2)} - 1\right)\right).$$

To facilitate comparisons with the numerical solutions, we determine the opening angle α as a function of the vertical displacement d of the beam's midpoint by determining the auxiliary angle β from

$$\tan\beta = \frac{2d}{L}.$$

The opening angle then follows from

$$\tan\left(\frac{\alpha}{2}\right) = \tan(\pi - 2\beta) = -\tan 2\beta = \frac{2\tan\beta}{\tan^2\beta - 1}.$$

Here is a comparison between the computed and analytically predicted values for the pressure p_{ext}, required to deflect the beam's midpoint by the specified displacement, d.

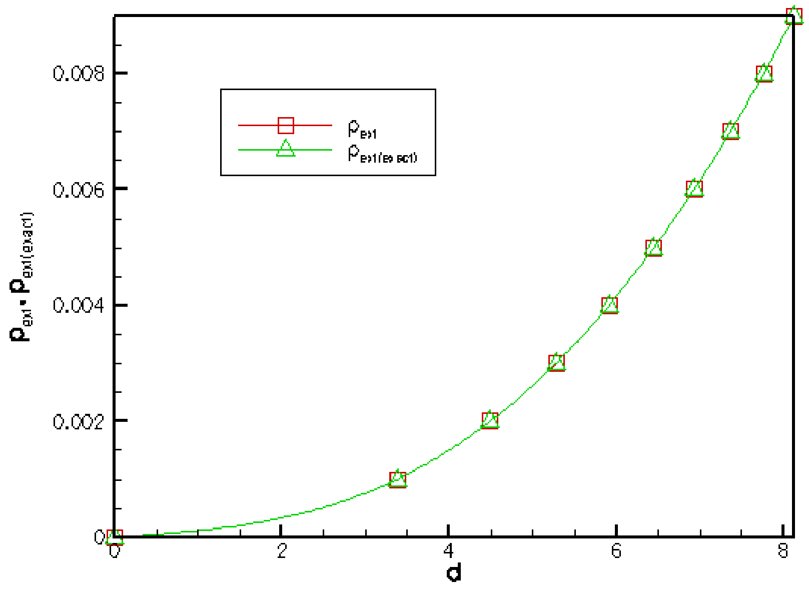

Figure 1.4 Comparison between the computed and analytical solutions.

Comment: Geometric and constitutive nonlinearities

The comparison between the computational results and the analytical predictions is very satisfying but is important to realise that the agreement only validates the numerical solution, not the physical model. `oomph-lib's HermiteBeamElement` is based on a **geometrically nonlinear** theory, implying that the kinematics of the deformation are captured exactly for arbitrarily large displacements and rotations. However, the use of **incrementally linear constitutive equations** in the variational principle (1) can only be justified if the strain is small. This is clearly not the case for the deformations shown above. The combination of geometric nonlinearity with linear constitutive equations can, however, be justified in applications in which the beam undergoes large displacements with little extension of its centreline. This typically occurs in stability problems, such as the buckling of a circular ring under external pressure, considered in another example.

1.4 Global parameters and functions

The namespace `Global_Physical_Variables` contains the dimensionless beam thickness, h, the dimensionless pre-stress, σ_0, and the pressure load, p_{ext}, as well as the function `Global_Physical_Variables::load()` which computes the load vector in the form required by the `HermiteBeamElements`. (The function's arguments allow the load vector to be a function of the Lagrangian and Eulerian coordinates, and the unit normal to the deformed beam, $\mathbf{N}$.)

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

```
{
 /// Non-dimensional thickness
 double H;

 /// 2nd Piola Kirchhoff pre-stress
 double Sigma0;

 /// Pressure load
 double P_ext;

 /// Load function: Apply a constant external pressure to the beam
 void load(const Vector<double>& xi, const Vector<double> &x,
           const Vector<double>& N, Vector<double>& load)
 {
  for(unsigned i=0;i<2;i++) {load[i] = -P_ext*N[i];}
 }
} // end of namespace
```

1.5 The driver code

The main code is very short. The physical parameters h, σ_0 and L are initialised and the problem is constructed using 10 elements. Following the usual self-test, we call the function ElasticBeamProblem::parameter_study() to compute the deformation of the beam for a range of external pressures.

```
//========start_of_main================================================
/// Driver for beam (string under tension) test problem
//=====================================================================
int main()
{
 // Set the non-dimensional thickness
 Global_Physical_Variables::H=0.01;

 // Set the 2nd Piola Kirchhoff prestress
 Global_Physical_Variables::Sigma0=0.1;

 // Set the length of domain
 double L = 10.0;
 // Number of elements (choose an even number if you want the control point
 // to be located at the centre of the beam)
 unsigned n_element = 10;
 // Construst the problem
 ElasticBeamProblem problem(n_element,L);
 // Check that we're ready to go:
 cout « "\n\n\nProblem self-test ";
 if (problem.self_test()==0)
  {
   cout « "passed: Problem can be solved." « std::endl;
  }
 else
  {
   throw OomphLibError("Self test failed",
                       OOMPH_CURRENT_FUNCTION,
                       OOMPH_EXCEPTION_LOCATION);
  }
 // Conduct parameter study
 problem.parameter_study();
} // end of main
```

1.6 The problem class

The problem class has five member functions, only two of which are non-trivial:

- the problem constructor, `ElasticBeamProblem(...)`, whose arguments specify the number of elements and the beam's undeformed length.
- the function `parameter_study()`, which computes the beam's deformation for a range of external pressures.

In the present problem, the functions `Problem::actions_before_newton_solve()` and `Problem↵ ::actions_after_newton_solve()` are not required, so remain empty. The function `ElasticBeamProblem::mesh_p` overloads the (virtual) function `Problem::mesh_pt()` to return a pointer to the specific mesh used in this problem. This avoids explicit re-casts when member functions of the specific mesh need to be accessed.
The class also includes three private data members which store a pointer to a node at which the displacement is documented, the length of the domain, and a pointer to the geometric object that specifies the beam's undeformed shape.

```
//======start_of_problem_class==========================================
/// Beam problem object
```

```
//=====================================================================
class ElasticBeamProblem : public Problem
{
public:

 /// Constructor: The arguments are the number of elements,
 /// the length of domain
 ElasticBeamProblem(const unsigned &n_elem, const double &length);

 /// Conduct a parameter study
 void parameter_study();

 /// Return pointer to the mesh
 OneDLagrangianMesh<HermiteBeamElement>* mesh_pt()
  {return dynamic_cast<OneDLagrangianMesh<HermiteBeamElement>*>
    (Problem::mesh_pt());}

 /// No actions need to be performed after a solve
 void actions_after_newton_solve() {}

 /// No actions need to be performed before a solve
 void actions_before_newton_solve() {}
private:

 /// Pointer to the node whose displacement is documented
 Node* Doc_node_pt;

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

 /// Pointer to geometric object that represents the beam's undeformed shape
 GeomObject* Undef_beam_pt;
}; // end of problem class
```

1.7 The Problem constructor

We start by creating the undeformed centreline of the beam as a `StraightLine`, one of `oomph-lib's` standard geometric objects.

```
//=============start_of_constructor=====================================
/// Constructor for elastic beam problem
//=====================================================================
ElasticBeamProblem::ElasticBeamProblem(const unsigned &n_elem,
                                        const double &length) : Length(length)
{
 // Set the undeformed beam to be a straight line at y=0
 Undef_beam_pt=new StraightLine(0.0);
```

We then construct the a one-dimensional Lagrangian mesh in two-dimensional space, using the previously-constructed geometric object to set the initial positions of the nodes.

```
 // Create the (Lagrangian!) mesh, using the geometric object
 // Undef_beam_pt to specify the initial (Eulerian) position of the
 // nodes.
 Problem::mesh_pt() =
  new OneDLagrangianMesh<HermiteBeamElement>(n_elem,length,Undef_beam_pt);
```

The `OneDLagrangianMesh` is a `SolidMesh` whose constituent nodes are `SolidNodes`. These nodes store not only their (variable) 2D Eulerian position, accessible via `SolidNode::x(...)`, but also their (fixed) 1D Lagrangian coordinates, accessible via `SolidNode::xi(...)`. The `OneDLagrangianMesh` constructor assigns the nodes' Lagrangian coordinate, ξ, by spacing them evenly in the range $\xi \in [0, L]$. The `GeomObject` pointed to by `Undef_beam_pt`, provides a parametrisation of the beam's undeformed shape in the form $\mathbf{r}(\xi)$, and this is used to determine the nodes's initial Eulerian position.

Next we pin the nodal positions on both boundaries

```
 // Set the boundary conditions: Each end of the beam is fixed in space
 // Loop over the boundaries (ends of the beam)
 for(unsigned b=0;b<2;b++)
  {
   // Pin displacements in both x and y directions
   // [Note: The mesh_pt() function has been overloaded
   //  to return a pointer to the actual mesh, rather than
   //  a pointer to the Mesh base class. The current mesh is derived
   //  from the SolidMesh class. In such meshes, all access functions
   //  to the nodes, such as boundary_node_pt(...), are overloaded
   //  to return pointers to SolidNodes (whose position can be
   //  pinned) rather than "normal" Nodes.]
   mesh_pt()->boundary_node_pt(b,0)->pin_position(0);
   mesh_pt()->boundary_node_pt(b,0)->pin_position(1);
  }
```

We then loop over the elements and set the pointers to the physical parameters (the pre-stress and the thickness), the function pointer to the load vector, and the pointer to the geometric object that specifies the undeformed beam shape.

```
 //Loop over the elements to set physical parameters etc.
 for(unsigned e=0;e<n_element;e++)
  {
   // Upcast to the specific element type
   HermiteBeamElement *elem_pt =
    dynamic_cast<HermiteBeamElement*>(mesh_pt()->element_pt(e));

   // Set physical parameters for each element:
   elem_pt->sigma0_pt() = &Global_Physical_Variables::Sigma0;
   elem_pt->h_pt() = &Global_Physical_Variables::H;
   // Set the load Vector for each element
   elem_pt->load_vector_fct_pt() = &Global_Physical_Variables::load;
   // Set the undeformed shape for each element
   elem_pt->undeformed_beam_pt() = Undef_beam_pt;
  } // end of loop over elements
```

We choose a node near the centre of the beam to monitor the displacements. (If the total number of nodes is even, the control node will not be located at the beam's exact centre; its vertical displacement will therefore differ from the analytical solution that we output in `doc_solution(...)` – in this case we issue a suitable warning.)

```
 // Choose node at which displacement is documented (halfway along -- provided
 // we have an odd number of nodes; complain if this is not the
 // case because the comparison with the exact solution will be wrong
 // otherwise!)
 unsigned n_nod=mesh_pt()->nnode();
 if (n_nod%2!=1)
  {
   cout « "Warning: Even number of nodes " « n_nod « std::endl;
   cout « "Comparison with exact solution will be misleading..." « std::endl;
  }
 Doc_node_pt=mesh_pt()->node_pt((n_nod+1)/2-1);
```

Finally, we assign the equation numbers

```
 // Assign the global and local equation numbers
 cout « "# of dofs " « assign_eqn_numbers() « std::endl;
} // end of constructor
```

1.8 The Parameter Study

The function ElasticBeamProblem::parameter_study() is used to perform a parameter study, computing the beam's deformation for a range of external pressures. During the solution of this particular problem, the maximum residual in the Newton iteration can be greater than the default maximum value of 10.0. We increase the default value by assigning a (much) larger value to `Problem::Max_residuals`.
Next, we choose the increment in the control parameter (the external pressure), set its initial value and open an output file that will contain the value of the external pressure, the mid-point displacement and external pressure computed from the analytical solution. We also create an output stream and a string that will be used to write the complete solution for each value of the external pressure.
In the loop, we increment the external pressure P_{ext}, solve the problem, calculate the analytical prediction for the pressure that is required to achieve the computed deformation, plot the solution and write the pressure, the displacement and exact pressure to the trace file.

```
//=======start_of_parameter_study==========================================
/// Solver loop to perform parameter study
//=========================================================================
void ElasticBeamProblem::parameter_study()
{
 // Over-ride the default maximum value for the residuals
 Problem::Max_residuals = 1.0e10;

 // Set the increments in control parameters
 double pext_increment = 0.001;

 // Set initial values for control parameters
 Global_Physical_Variables::P_ext = 0.0 - pext_increment;

 // Create label for output
 DocInfo doc_info;

 // Set output directory -- this function checks if the output
 // directory exists and issues a warning if it doesn't.
 doc_info.set_directory("RESLT");

 // Open a trace file
 ofstream trace("RESLT/trace_beam.dat");

 // Write a header for the trace file
 trace «
  "VARIABLES=\"p_e_x_t\",\"d\"" «
  ", \"p_e_x_t_(_e_x_a_c_t_)\"" « std::endl;

 // Output file stream used for writing results
 ofstream file;
 // String used for the filename
 char filename[100];
 // Loop over parameter increments
 unsigned nstep=10;
 for(unsigned i=1;i<=nstep;i++)
  {
   // Increment pressure
   Global_Physical_Variables::P_ext += pext_increment;

   // Solve the system
   newton_solve();

   // Calculate exact solution for `string under tension' (applicable for
   // small wall thickness and pinned ends)
   // The tangent of the angle beta
   double tanbeta =-2.0*Doc_node_pt->x(1)/Length;
   double exact_pressure = 0.0;
   //If the beam has deformed, calculate the pressure required
   if(tanbeta!=0)
    {

      //Calculate the opening angle alpha
      double alpha = 2.0*atan(2.0*tanbeta/(1.0-tanbeta*tanbeta));
      // Jump back onto the main branch if alpha>180 degrees
      if (alpha<0) alpha+=2.0*MathematicalConstants::Pi;
     // Green strain:
     double gamma=0.5*(0.25*alpha*alpha/(sin(0.5*alpha)*sin(0.5*alpha))-1.0);
     //Calculate the exact pressure
     exact_pressure=Global_Physical_Variables::H*
      (Global_Physical_Variables::Sigma0+gamma)*alpha/Length;
    }

   // Document the solution
   sprintf(filename,"RESLT/beam%i.dat",i);
   file.open(filename);
   mesh_pt()->output(file,5);
   file.close();

   // Write trace file: Pressure, displacement and exact solution
   // (for string under tension)
   trace « Global_Physical_Variables::P_ext  « " "
         « abs(Doc_node_pt->x(1))
         « " " « exact_pressure
         « std::endl;
  }

} // end of parameter study
```

1.9 Exercises and Comments

1. Modify the code so that one end of the beam is no longer fixed in space. What happens? Why?
2. Increase the bending effects by increasing the beam's thickness to $h = 0.1$, say, and by "clamping" its ends, so that $d(\mathbf{R}_w(\xi) \cdot \mathbf{e}_y)/d\xi = 0$ at $\xi = 0$ and $\xi = L$. This condition can be enforced by pinning the "type 1" (slope) positional degree of freedom in the vertical (1) direction at both ends; this requires the insertion of the statement
   ```
   mesh_pt()->boundary_node_pt(b,0)->pin_position(1,1);
   ```
   in the loop in the `Problem` constructor. You will have to adjust the number of elements to fully resolve the bending boundary layers.
3. `oomph-lib's` defaultnonlinear solver, `Problem::newton_solve(...)`, provides an implementation of the Newton method, which has the attractive feature that it converges quadratically – provided a good initial guess for the solution is available. Good initial guesses can often (usually?) be generated by computing the solution via a sequence of substeps, as in the above example where we started with a known solution (the undeformed beam – the exact solution for $p_{ext} = p_{ext}^{(0)} = 0$) and used it as the initial guess for the solution at a small external pressure, $p_{ext}^{(1)}$. When the Newton method converged, we used the computed solution as the initial guess for the solution at a slightly larger pressure, $p_{ext}^{(2)}$, etc. If the increase in the load (or some other control parameter) is too large, the Newton method will diverge. To avoid unnecessary computations, the Newton iteration is terminated if:
   - the number of iterations exceeds `Problem::Max_newton_iterations` (which has a default value of 10)
   - the residual exceeds `Problem::Max_residuals` (which has a default value of 10.0)

The Newton method continues until the maximum residual has been reduced to `Problem::Newton_solver↵_tolerance`, which has a default value of 10^{-8}.

All three values are protected data members of the `Problem` base class and can therefore be changed in any specific `Problem`. For instance, in the problem considered above, the undeformed beam provides a poor approximation of its equilibrium shape at the first pressure value. The Newton method still converges (very slowly initially, then quadratically as it approaches the exact solution), even though the initial maximum residual has a relatively large value of 19.6. Here are some exercises that explore the convergence characteristics of the Newton method:

1. Experiment with the Newton solver and find the largest value for the load increment, `pext_increment`, for which the Newton method still converges.
2. Explain why the Newton method converges very slowly for small values of p_{ext} and much more rapidly at larger values, even though the load increment, `pext_increment`, remains constant.
3. Compare the solutions for different values of `Problem::Newton_solver_tolerance`. Is the default value
10^{-8} adequate? Reduce it to 10^{-12} and 10^{-18}. What do you observe?

1.10 Source files for this tutorial

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

  `demo_drivers/beam/tensioned_string/`

- The driver code is:

  `demo_drivers/beam/tensioned_string/tensioned_string.cc`

1.11 PDF file

A `pdf version` of this document is available.