Chapter 1

The equations of time-harmonic linear elasticity and use of PMLs

The aim of this tutorial is to demonstrate the solution of the time-harmonic equations of linear elasticity in Cartesian coordinates. These equations are useful to describe forced, time-harmonic oscillations of elastic bodies. The implementation extends the standard formulation discussed in another tutorial, which is modified to incorporate perfectly matched layers (PMLs) as an alternative to classical absorbing/approximate boundary conditions or DtN maps.

We first present details on the relevant theory and its extensions and proceed with an example problem of time-harmonic oscillations on an infinite domain with an interior circular boundary.

Acknowledgement: The implementation of the equations and the documentation were developed jointly with Radu Cimpeanu (Imperial College London), with financial support from Thales Underwater.

1.1 Theory

Consider an infinite, linearly elastic body (of density ρ, Young's modulus E and Poisson's ratio ν), occupying the region D whose inner boundary is ∂D_d. Assuming that the body performs time-harmonic oscillations of frequency of ω its motion is governed by the equations of time-harmonic linear elasticity

$$\boldsymbol{\nabla}^* \cdot \boldsymbol{\tau}^* + \rho \boldsymbol{F}^* = -\rho\omega^2 \boldsymbol{u}^*,$$

where the x_i^* are the Cartesian coordinates, and the time-periodic stresses, body force and displacements are given by $\mathrm{Re}\{\boldsymbol{\tau^*}(x_i^*)\mathrm{e}^{-\mathrm{i}\omega t^*}\}$, $\mathrm{Re}\{\mathbf{F}^*(x_i^*)\mathrm{e}^{-\mathrm{i}\omega t^*}\}$ and $\mathrm{Re}\{\mathbf{u}^*(x_i^*)\mathrm{e}^{-\mathrm{i}\omega t^*}\}$ respectively. Note that, as usual, the superscript asterisk notation is used to distinguish dimensional quantities from their non-dimensional counterparts where required.
The body is subject to imposed time-harmonic displacements $\mathrm{Re}\{\hat{\mathbf{u}}^*\mathrm{e}^{-\mathrm{i}\omega t^*}\}$ along ∂D_d. This requires that

$$\mathbf{u}^* = \hat{\mathbf{u}}^* \text{ on } \partial D_d.$$

The stresses and displacements are related by the constitutive equations

$$\boldsymbol{\tau}^* = \frac{E}{1+\nu}\left(\frac{\nu}{1-2\nu}(\boldsymbol{\nabla}^* \cdot \mathbf{u}^*)\mathbf{I} + \frac{1}{2}(\boldsymbol{\nabla}^*\mathbf{u}^* + (\boldsymbol{\nabla}^*\mathbf{u}^*)\mathrm{T})\right),$$

where $(\boldsymbol{\nabla}^*\mathbf{u}^*)^{\mathrm{T}}$ represents the transpose of $\boldsymbol{\nabla}^*\mathbf{u}^*$.

We non-dimensionalise the equations, using a problem specific reference length, $\mathcal{L}$, and a timescale $\mathcal{T} = \frac{1}{\omega}$, and use Young's modulus to non-dimensionalise the body force and the stress tensor:

$$\boldsymbol{\tau}^* = E\,\boldsymbol{\tau}, \qquad x_i^* = \mathcal{L}\,x_i$$
$$\mathbf{u}^* = \mathcal{L}\,\mathbf{u}, \qquad \mathbf{F}^* = \frac{E}{\rho\mathcal{L}}\,\mathbf{F}, \qquad t^* = \mathcal{T}\,t.$$

The non-dimensional form of the equations is then given by

$$\boldsymbol{\nabla}\cdot\boldsymbol{\tau} + \mathbf{F} = -\Omega^2\mathbf{u}, \qquad (1)$$

with the non-dimensional constitutive relation,

$$\boldsymbol{\tau} = \frac{1}{1+\nu}\left(\frac{\nu}{1-2\nu}(\boldsymbol{\nabla}\cdot\mathbf{u})\mathbf{I} + \frac{1}{2}(\boldsymbol{\nabla}\mathbf{u} + (\boldsymbol{\nabla}\mathbf{u})^{\mathrm{T}})\right). \qquad (2)$$

The non-dimensional parameter

$$\Omega = \mathcal{L}\omega\sqrt{\frac{\rho}{E}}$$

is the ratio of the elastic body's intrinsic timescale, $\mathcal{L}\sqrt{\frac{\rho}{E}}$, to the problem-specific timescale, $\mathcal{T} = \frac{1}{\omega}$, that we used to non-dimensionalise time. Ω can be interpreted as a non-dimensional version of the excitation frequency; alternatively/equivalently Ω^2 may be interpreted as a non-dimensional density. The boundary conditions are set as

$$\mathbf{u} = \hat{\mathbf{u}} \text{ on } \partial D_d.$$

1.2 Perfectly matched layers

The idea behind perfectly matched layers is illustrated in the figure below. The actual physical/mathematical problem has to be solved in the infinite domain D (shown on the left), with a boundary condition at a certain distance from the source of the waves ensuring the suitable decay of the solution at large distances from the region of interest.
If computations are performed in a finite computational domain, D_c , (shown in the middle), spurious wave reflections are likely to be generated at the artificial boundary ∂D_c of the computational domain.
The idea behind PML methods is to surround the actual computational domain D_c with a layer of "absorbing" material whose properties are chosen such that the outgoing waves are absorbed within it, without creating any artificial reflected waves at the interface between the PML layer and the computational domain.

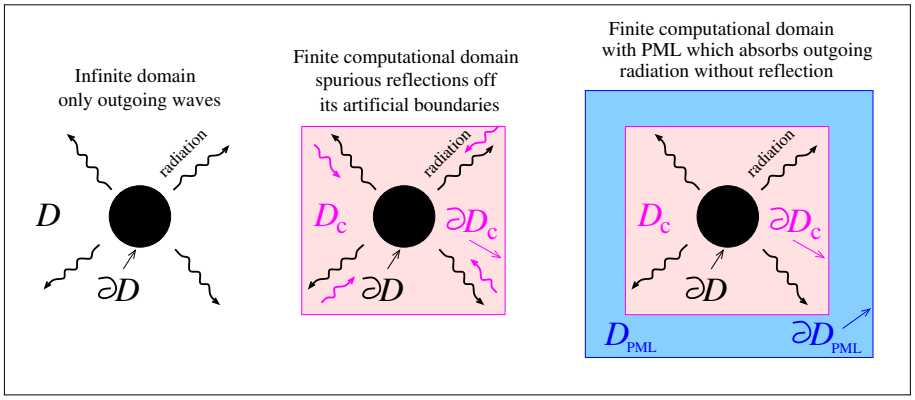

Figure 1.1 Sketch illustrating the idea behind perfectly matched layers.

Our implementation of the perfectly matched layers follows the development in A. Bermudez, L. Hervella-Nieto, A. Prieto, and R. Rodriguez "An optimal perfectly matched layer with unbounded absorbing function for time-harmonic acoustic scattering problems" Journal of Computational Physics **223** 469-488 (2007) and we assume the boundaries of the computational domain to be aligned with the coordinate axes, as shown in the sketch above.
The method requires a slight generalisation of the equations, achieved by introducing the complex coordinate mapping

$$\frac{\partial}{\partial x_j} \rightarrow \frac{1}{\gamma_j}\frac{\partial}{\partial x_j} \qquad (3)$$

within the perfectly matched layers.
The choice of γ_j depends on the orientation of the PML layer. Since we are restricting ourselves to axis-aligned mesh boundaries we need to distinguish three different cases, as shown in the sketch below:

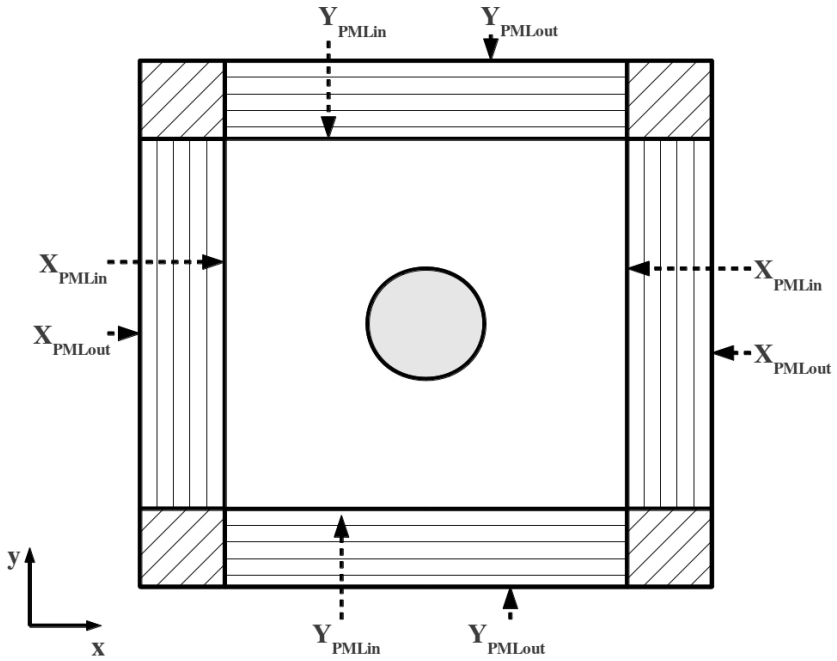

Figure 1.2 Sketch illustrating the geometry of the perfectly matched layers.

We follow the work of U. Basu and A. Chopra "Perfectly matched layers for time-harmonic elastodynamics of unbounded domains: theory and finite-element implementation", Computer Methods in Applied Mechanics and Engineering **192** 1337-1375(2003) and choose:

- For layers that are aligned with the y axis (such as the left and right PML layers in the sketch, with vertical stripes) we set

$$\gamma_x(x) = 1 + i\frac{f_0}{a_0}\,\sigma_x(x) \qquad \text{with} \quad \sigma_x(x) = \frac{1}{|X_{PML_{out}} - x|} - \frac{1}{|X_{PML_{out}} - X_{PML_{in}}|}, \qquad (4)$$

  where $a_0 = \sqrt{2.0(1.0+\nu)}\Omega$, f_0 is a constant to be tuned, $X_{PML_{in}}$ is the x-coordinate of the inner boundary of the PML layer (the coordinate of the interface with the physical domain), $X_{PML_{out}}$ is the x-coordinate of the outer boundary of the PML layer , and

$$\gamma_y = 1.$$

- For layers that are aligned with the x axis (such as the top and bottom PML layers in the sketch, with horizontal stripes) we set

$$\gamma_x = 1,$$

  and

$$\gamma_y(y) = 1 + i\frac{f_0}{a_0}\,\sigma_y(y) \qquad \text{with} \quad \sigma_y(y) = \frac{1}{|Y_{PML_{out}} - y|} - \frac{1}{|Y_{PML_{out}} - Y_{PML_{in}}|}, \qquad (5)$$

where Y_{PML} is the y-coordinate of the outer boundary of the PML layer.

- In corner regions (diagonally striped) that are bounded by two axis-aligned PML layers (with outer coordinates X_{PML} and Y_{PML}) we set

$$\gamma_x(x) = 1 + i\frac{f_0}{a_0}\,\sigma_x(x) \qquad \text{with} \quad \sigma_x(x) = \frac{1}{|X_{PML_{out}} - x|} - \frac{1}{|X_{PML_{out}} - X_{PML_{in}}|} \tag{6}$$

and

$$\gamma_y(y) = 1 + i\frac{f_0}{a_0}\,\sigma_y(y) \qquad \text{with} \quad \sigma_y(y) = \frac{1}{|Y_{PML_{out}} - y|} - \frac{1}{|Y_{PML_{out}} - Y_{PML_{in}}|}. \tag{7}$$

- Finally, in the actual computational domain (outside the PML layers) we set

$$\gamma_x(x) = \gamma_y(y) = 1.$$

In our numerical experiments $f_0 = 20.0$ was found to be an appropriate value, which gives acceptable numerical behaviour. This is well aligned with the cited paper, where this parameter ranges from roughly 10.0 to 50.0.

1.2.1 Implementation of the perfectly matched layers within oomph-lib

The finite-element-discretised equations (modified by the PML terms discussed above) are implemented in the `PML↩TimeHarmonicLinearElasticityEquations<DIM>` class which is templated by the spatial dimension, `DIM`. As usual, we provide fully functional elements by combining these with geometric finite elements (from the Q and T families – corresponding (in 2D) to triangles and quad elements). By default, the PML modifications are disabled, i.e. $\gamma_x(x)$ and $\gamma_y(y)$ are both set to 1.

The generation of suitable 2D PML meshes along the axis-aligned boundaries of a given bulk mesh is facilitated by helper functions which automatically erect layers of (quadrilateral) PML elements. The layers are built from `QPMLTimeHarmonicLinearElasticityElement<2,NNODE_1D>` elements and the parameter `NNO↩DE_1D` is automatically chosen to match that of the elements in the bulk mesh. The bulk mesh can contain quads or triangles (as shown in the specific example presented below).

For instance, to erect a PML layer (of width `width`, with `n_pml` elements across the width of the layer) on the "right" boundary (with boundary ID `b_bulk`) of the bulk mesh pointed to by `bulk_mesh_pt`, a call to

```
TwoDimensionalPMLHelper::create_right_pml_mesh
<PMLLayerElement<ELASTICITY_ELEMENT> >
(Solid_mesh_pt, right_boundary_id,
Global_Parameters::n_x_right_pml,
Global_Parameters::width_x_right_pml);
```

returns a pointer to a newly-created mesh that contains the PML elements which are automatically attached to the boundary of the bulk mesh (i.e. the `Nodes` on the outer boundary of the bulk mesh are shared (pointed to), rather than duplicated, by the elements in the PML mesh). The PML-ness of the elements is automatically enabled, i.e. the functions $\gamma_x(x)$ and $\gamma_y(y)$ are set as described above. Finally, zero Dirichlet boundary conditions are applied to the real and imaginary parts of the solution on the outer boundary of the PML layer.

Similar helper functions exist for PML layers on other axis-aligned boundaries, and for corner PML meshes; see the code listings provided below. Currently, we only provide this functionality for convex 2D computational domains, but the generalisation to non-convex boundaries and 3D is straightforward (if tedious) to implement (Any volunteers?).

1.3 Implementation

Within `oomph-lib`, the non-dimensional version of equations (1) with the constitutive equations (2) are implemented in the `PMLTimeHarmonicLinearElasticityEquations<DIM>` equations class, where the template parameter `DIM` indicates the spatial dimension. Following our usual approach, discussed in the `(Not-So-)Quick Guide`, this equation class is then combined with a geometric finite element to form a fully-functional finite element. For instance, the combination of the `PMLTimeHarmonicLinearElasticity↩Equations<2>` class with the geometric finite element `QElement<2,3>` yields a nine-node quadrilateral element. As usual, the mapping between local and global (Eulerian) coordinates within an element is given by,

$$x_i = \sum_{j=1}^{N^{(E)}} X_{ij}^{(E)}\,\psi_j, \qquad i = 1, 2,$$

where $N^{(E)}$ is the number of nodes in the element, $X_{ij}^{(E)}$ is the i-th global (Eulerian) coordinate of the j-th `Node` in the element, and the ψ_j are the element's shape functions, defined in the geometric finite element.
All the constitutive parameters are real. The two components of the displacement field have a real and imaginary part. We store the four real-valued nodal unknowns in the order $\mathrm{Re}\{u_x\}, \mathrm{Re}\{u_y\}, \mathrm{Im}\{u_x\}, \mathrm{Im}\{u_y\}$ and use the shape functions to interpolate the displacements as

$$u_i^{(n)} = \sum_{j=1}^{N^{(E)}} U_{ij}^{(E)} \psi_j, \qquad i = 1, ...4,$$

where $U_{ij}^{(E)}$ is the i-th displacement component (enumerated as described above) at the j-th `Node` in the element.

1.4 A specific example: outward propagation of elastic waves from the surface of a cylindrical object

We consider the time-harmonic deformation of a 2D elastic body that occupies the region outside a circle whose diameter we use as the lengthscale $\mathcal{L}$. We impose a displacement $\mathbf{u}(r, \theta) = u_0 \mathbf{e}_r + u_1 \mathbf{e}_\theta$ on the inner boundary (to generate a combination of pressure waves and shear waves) and allow for a formulation with perfectly matched layers on the outer boundary of the computational domain, which should allow for a smooth propagation of the elastic waves with no interference from the restriction of the problem to a finite computational domain. $\mathbf{e}_r$ is the unit vector in the radial direction, whereas $\mathbf{e}_\theta$ is the unit vector in azimuthal direction.
It is easy to find an analytical solution of this problem by working in polar coordinates and exploiting the axisymmetry of the solution by writing the displacements as $\mathbf{u} = U(r)\mathbf{e}_r$ or $\mathbf{u} = U(r)\mathbf{e}_\theta$. The displacement $U(r)$ is then governed in both cases by an equation of type

$$\frac{d}{dr}\left(\frac{U}{r} + \frac{dU}{dr}\right) + k^2 U = 0,$$

where $k^2 = k_r^2 = \dfrac{\Omega^2}{\lambda + 2\mu}$ for the radial component and $k^2 = k_\theta^2 = \dfrac{\Omega^2}{\mu}$ for the azimuthal component. We also note that

$$\lambda = \frac{\nu}{(1+\nu)(1-2\nu)} \qquad \text{and} \qquad \mu = \frac{1}{2(1+\nu)}$$

are the non-dimensional Lame parameters. The solution of these equations is given in each case by:

$$U(r) = H_1(kr).$$

where H_1 is the Hankel function of first kind. The two solutions can then be converted to Cartesian coordinates and added together in an appropriate manner. The details of the procedure can be found in the `GlobalParameters::exact_u()` function.
We note that even though a relatively simple analytical solution (in polar coordinates!) exists for this problem, it is a non-trivial test case for our code which solves the governing equations in Cartesian coordinates. It is also a highly relevant test case in the context of testing perfectly matched layers, since the solution contains both shear and compression waves.

1.5 Results

The discretised geometry is shown below. We choose a circle of radius $r = 0.5$ in a computational domain otherwise set to $-2 \leq x \leq 2, -2 \leq y \leq 2$, with a PML thickness of 1.6 in each dimension, spanned by 16 elements.

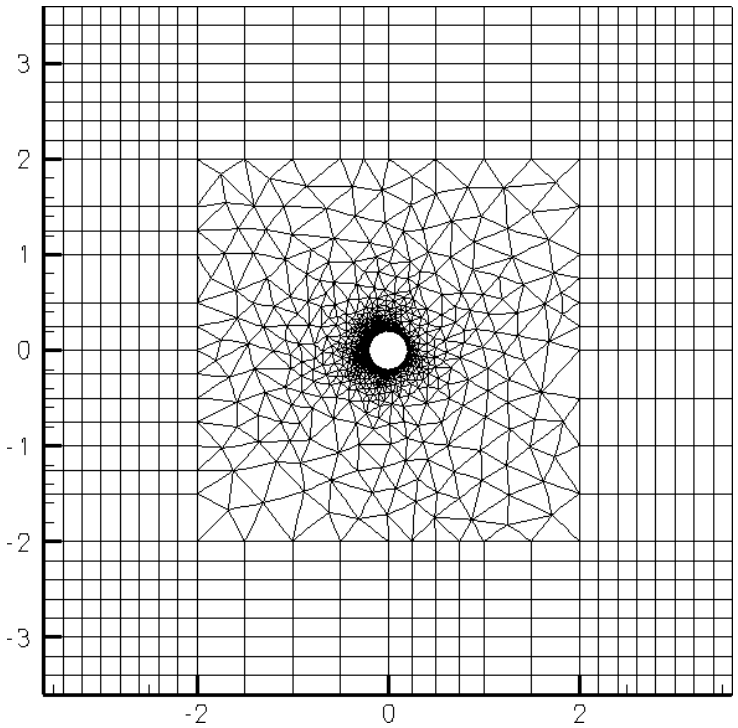

Figure 1.3 Mesh used to resolve the geometry.

The figures below show "carpet plots" (on the left) of the real and imaginary parts of the exact (green) and computed (blue) horizontal displacement for $\Omega^2 = 30.0$ and $\nu = 0.3$. Comparing the two solutions shows good agreement and hence an efficient damping given by the perfectly matched layers. We can also observe this by taking a one-dimensional slice from point $(-2.0, 0.0)$ to point $(2.0, 0.0)$ and visualising the profiles of the exact and computed solutions, shown on the right. The green line represents the profile of the exact solution, whereas the blue dots are extracted from the profile of the computed solution.

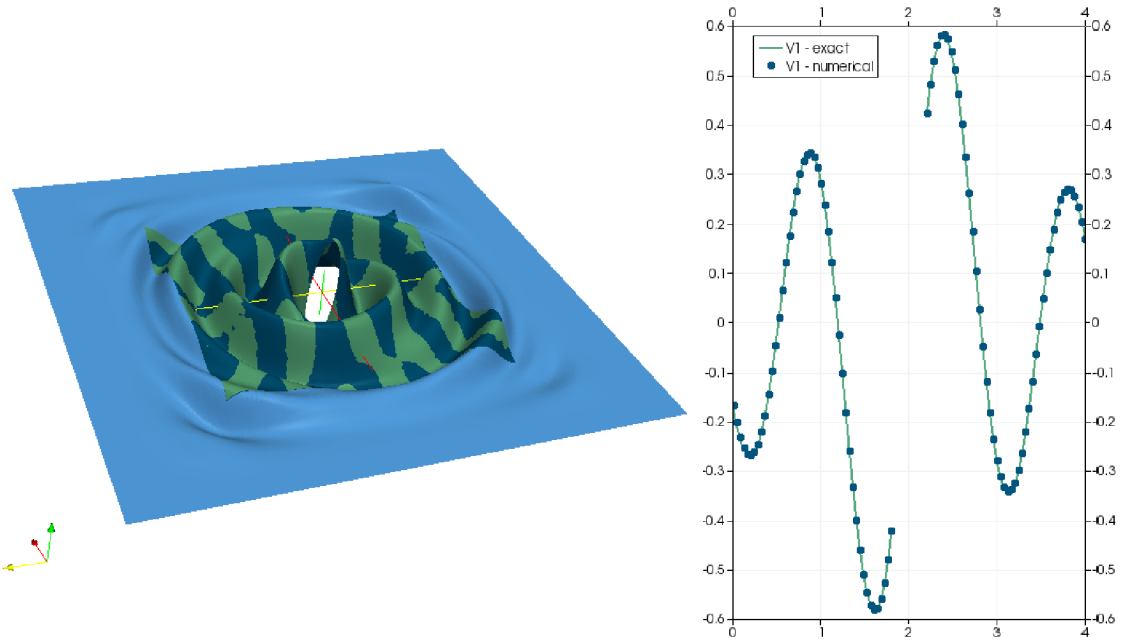

Figure 1.4 Real part of the horizontal displacement. Exact and computed solution carpet plot (left) and (-2.0,0.0) to (2.0,0.0) one-dimensional slice (right). Green: exact; dark blue: computed; light blue: perfectly matched layers.

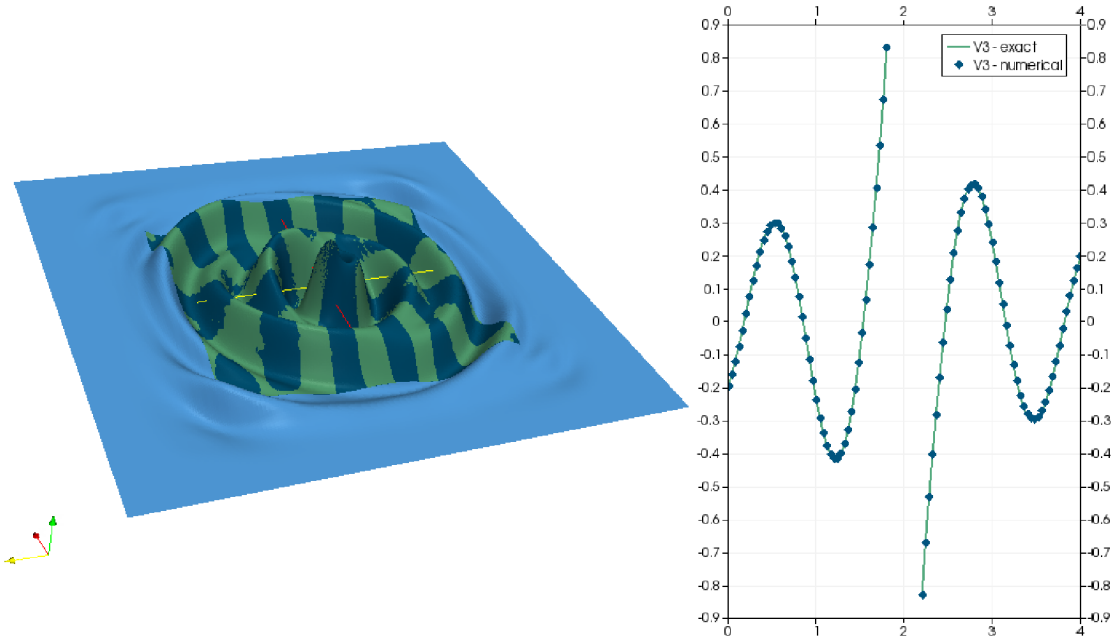

Figure 1.5 Imaginary part of the horizontal displacement. Exact and computed solution carpet plot (left) and (-2.0,0.0) to (2.0,0.0) one-dimensional slice (right). Green: exact; dark blue: computed; light blue: perfectly matched layers.

To demonstrate that the resulting displacement field is indeed axisymmetric, we present plots of the real and imaginary parts of the radial displacement, $(Re(u_1)^2 + Re(u_2)^2)^{1/2}$ and $(Im(u_1)^2 + Im(u_2)^2)^{1/2}$.

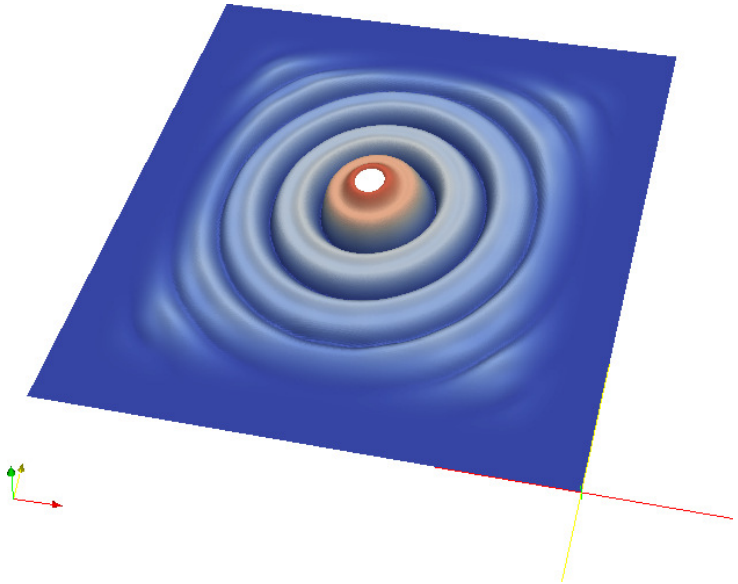

Figure 1.6 Real part of the computed radial displacement.

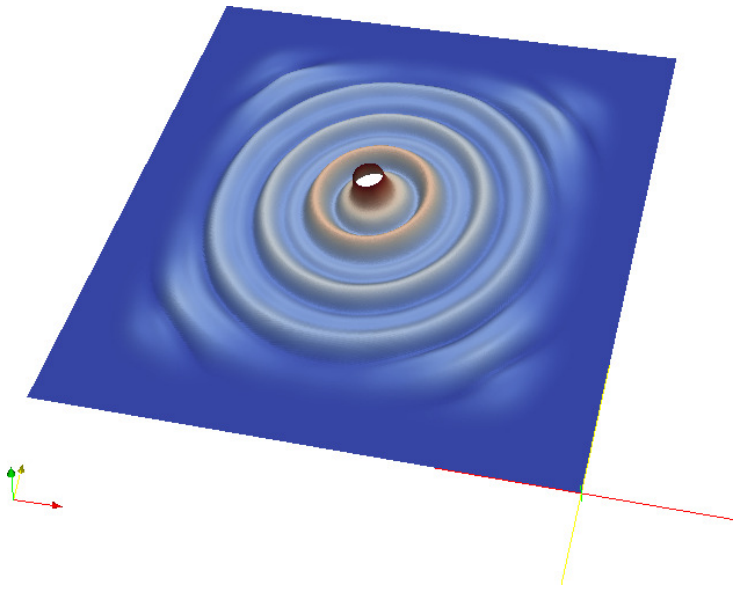

Figure 1.7 Imaginary part of the computed radial displacement.

1.6 Global parameters and functions

As usual, we define all non-dimensional parameters in a namespace where we also define the exact solution which is imposed at the inner boundary. We omit the (lengthy) listing of the exact solution.

```
//=======start_namespace==========================================
/// Global variables
//================================================================
namespace Global_Parameters
{
 /// helper to set target mesh element size
 double Element_area = 0.01;

 /// helpers to time the code
 double T_start = 0.0;
 double T_end = 0.0;

 /// PML width in elements for the right layer
 unsigned N_pml_multiplier = 1;
 double L_pml_multiplier = 1.0;

 /// PML width in elements for the right layer
 unsigned N_x_right_pml = 8;

 /// PML width in elements for the top layer
 unsigned N_y_top_pml = 8;

 /// PML width in elements for the left layer
 unsigned N_x_left_pml = 8;

 /// PML width in elements for the left layer
 unsigned N_y_bottom_pml = 8;
 // Outer physical length of the PML layers
 // defaults to 0.2, so 10% of the size of the
 // physical domain
 double Width_x_right_pml  = 2.0;
 double Width_y_top_pml    = 2.0;
 double Width_x_left_pml   = 2.0;
 double Width_y_bottom_pml = 2.0;

 /// Function to compute dependent parameters
 void compute_dependent_parameters()
 {
  /// Adjust number of PML elements, set to be equal for all layers
  N_x_right_pml  = N_x_right_pml  * N_pml_multiplier;
  N_x_left_pml   = N_x_left_pml   * N_pml_multiplier;
  N_y_top_pml    = N_y_top_pml    * N_pml_multiplier;
  N_y_bottom_pml = N_y_bottom_pml * N_pml_multiplier;

  ///Adjust physical size of PML layers, set to be equal for all layers
  Width_x_right_pml  = Width_x_right_pml  * L_pml_multiplier;
  Width_x_left_pml   = Width_x_left_pml   * L_pml_multiplier;
```

```
 Width_y_top_pml    = Width_y_top_pml    * L_pml_multiplier;
 Width_y_bottom_pml = Width_y_bottom_pml * L_pml_multiplier;
}

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

/// Square of non-dim frequency
double Omega_sq=30.0;

/// The elasticity tensor
PMLTimeHarmonicIsotropicElasticityTensor* E_pt;

/// Thickness of annulus
double H_annulus=0.5;

/// Output directory
string Directory="RESLT";
```

1.7 The driver code

We start by reading the command line arguments, which allow the specification of the perfectly matched layers in terms of number of elements and thickness.

```
//=======start_of_main====================================================
/// Driver for annular disk loaded by pressure
//========================================================================
int main(int argc, char **argv)
{
 // Start timing of the code
 Global_Parameters::T_start=TimingHelpers::timer();
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);
 // Define possible command line arguments and parse the ones that
 // were actually specified
 // Over-write PML layers element number in each dimension
 CommandLineArgs::specify_command_line_flag("--n_pml",
                &Global_Parameters::N_pml_multiplier);
 // Over-write PML layers physical length in each dimension
 CommandLineArgs::specify_command_line_flag("--l_pml",
                &Global_Parameters::L_pml_multiplier);
 // Output directory
 CommandLineArgs::specify_command_line_flag(
  "--dir", &Global_Parameters::Directory);
```

After extracting the relevant information from the command line we continue by computing the dependent parameters.

```
 // Validation run?
 CommandLineArgs::specify_command_line_flag("--validation");
 // Parse command line
 CommandLineArgs::parse_and_assign();

 // Doc what has actually been specified on the command line
 CommandLineArgs::doc_specified_flags();
 // Validation run?
 if (CommandLineArgs::command_line_flag_has_been_set("--validation"))
  {
   oomph_info << "Using coarser resolution for self-test\n";

   /// Number of elements for each layer
   Global_Parameters::N_x_right_pml  = 2;
   Global_Parameters::N_y_top_pml    = 2;
   Global_Parameters::N_x_left_pml   = 2;
   Global_Parameters::N_y_bottom_pml = 2;

   /// Thickness of each layer
   Global_Parameters::Width_x_right_pml  = 1.0;
   Global_Parameters::Width_y_top_pml    = 1.0;
   Global_Parameters::Width_x_left_pml   = 1.0;
   Global_Parameters::Width_y_bottom_pml = 1.0;

   /// Target element size
   Global_Parameters::Element_area  = 0.025;

   /// Target element size
   Global_Parameters::Omega_sq  = 10.0;
  }

 /// Update dependent parameters
 Global_Parameters::compute_dependent_parameters();
 DocInfo doc_info;

 // Set output directory
 doc_info.set_directory(Global_Parameters::Directory);
```

Next, we create the elasticity tensor and set up the problem (discretised with six-noded triangular elements).

```
 // Build elasticity tensor
 Global_Parameters::E_pt=new PMLTimeHarmonicIsotropicElasticityTensor(
    Global_Parameters::Nu);
#ifdef ADAPTIVE
 //Set up the problem
 ElasticAnnulusProblem<
 ProjectablePMLTimeHarmonicLinearElasticityElement
  <TPMLTimeHarmonicLinearElasticityElement<2,3> >
 > problem;
#else
 //Set up the problem
 ElasticAnnulusProblem<TPMLTimeHarmonicLinearElasticityElement<2,3> >
  problem;
#endif
```

and perform the actual computation and the post-processing of the results:

```
 // Solve the problem using Newton's method
 problem.newton_solve();
 // End timing of the code
 Global_Parameters::T_end=TimingHelpers::timer();
 // Doc solution
 problem.doc_solution(doc_info);

} //end of main
```

1.8 The problem class

The `Problem` class is very simple and is very similar to that employed for the `solution of the classical time harmonic linear elasticity problem with traction boundary conditions.` We provide helper functions to create the PML meshes and to apply the boundary conditions (mainly because these tasks have to be performed repeatedly in the spatially adaptive version this code which is not discussed explicitly here; but see Comments and Exercises).

```
//============begin_problem============================================
/// Annular disk
//======================================================================
template<class ELASTICITY_ELEMENT>
class ElasticAnnulusProblem : public Problem
{
public:

 /// Constructor:
 ElasticAnnulusProblem();

 /// Destructor (empty)
 ~ElasticAnnulusProblem(){}

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

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

 /// Create PML meshes
 void create_pml_meshes();

 /// Actions before adapt: Wipe the mesh of traction elements
 void actions_before_adapt();

 /// Actions after adapt: Rebuild the mesh of traction elements
 void actions_after_adapt();

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

The private member data includes pointers the bulk mesh

```
 /// Pointer to solid mesh
 TriangleMesh<ELASTICITY_ELEMENT>* Solid_mesh_pt;
```

and to the various PML sub-meshes

```
 /// Pointer to the right PML mesh
 Mesh* PML_right_mesh_pt;

 /// Pointer to the top PML mesh
 Mesh* PML_top_mesh_pt;

 /// Pointer to the left PML mesh
 Mesh* PML_left_mesh_pt;

 /// Pointer to the bottom PML mesh
 Mesh* PML_bottom_mesh_pt;

 /// Pointer to the top right corner PML mesh
 Mesh* PML_top_right_mesh_pt;
```

```
 /// Pointer to the top left corner PML mesh
 Mesh* PML_top_left_mesh_pt;

 /// Pointer to the bottom right corner PML mesh
 Mesh* PML_bottom_right_mesh_pt;

 /// Pointer to the bottom left corner PML mesh
 Mesh* PML_bottom_left_mesh_pt;

 /// DocInfo object for output
 DocInfo Doc_info;

 /// Boundary ID of upper inner boundary
 unsigned Upper_inner_boundary_id;

 /// Boundary ID of upper outer boundary
 unsigned Upper_outer_boundary_id;

 /// Boundary ID of lower inner boundary
 unsigned Lower_inner_boundary_id;

 /// Boundary ID of lower outer boundary
 unsigned Lower_outer_boundary_id;

};
```

1.9 The problem constructor

We start by creating the `Circle` object that defines the inner boundary of the domain. The radius has been defined as part of the `Global_Parameters` definition:

```
//===========start_of_constructor=======================================
/// Constructor:
//======================================================================
template<class ELASTICITY_ELEMENT>
ElasticAnnulusProblem<ELASTICITY_ELEMENT>::ElasticAnnulusProblem()
{

 // Solid mesh
 //-----------
 // Create circle representing inner boundary
 double a=0.2;
 double x_c=0.0;
 double y_c=0.0;
 Circle* inner_circle_pt=new Circle(x_c,y_c,a);
```

and define the polygonal outer boundary of the computational domain.

```
 // Outer boundary
 //---------------
 TriangleMeshClosedCurve* outer_boundary_pt=0;

 unsigned n_segments = 16;
 Vector<TriangleMeshCurveSection*> outer_boundary_line_pt(4);

 // Each polyline only has three vertices, provide storage for their
 // coordinates
 Vector<Vector<double> > vertex_coord(2);
 for(unsigned i=0;i<2;i++)
  {
   vertex_coord[i].resize(2);
  }
 // First polyline
 vertex_coord[0][0]=-2.0;
 vertex_coord[0][1]=-2.0;
 vertex_coord[1][0]=-2.0;
 vertex_coord[1][1]=2.0;

 // Build the 1st boundary polyline
 unsigned boundary_id=2;
 outer_boundary_line_pt[0] = new TriangleMeshPolyLine(vertex_coord,
                                                      boundary_id);

 // Second boundary polyline
 vertex_coord[0][0]=-2.0;
 vertex_coord[0][1]=2.0;
 vertex_coord[1][0]=2.0;
 vertex_coord[1][1]=2.0;

 // Build the 2nd boundary polyline
 boundary_id=3;
 outer_boundary_line_pt[1] = new TriangleMeshPolyLine(vertex_coord,
                                                      boundary_id);

 // Third boundary polyline
 vertex_coord[0][0]=2.0;
 vertex_coord[0][1]=2.0;
```

```
vertex_coord[1][0]=2.0;
vertex_coord[1][1]=-2.0;

// Build the 3rd boundary polyline
boundary_id=4;
outer_boundary_line_pt[2] = new TriangleMeshPolyLine(vertex_coord,
                                                     boundary_id);

// Fourth boundary polyline
vertex_coord[0][0]=2.0;
vertex_coord[0][1]=-2.0;
vertex_coord[1][0]=-2.0;
vertex_coord[1][1]=-2.0;

// Build the 4th boundary polyline
boundary_id=5;
outer_boundary_line_pt[3] = new TriangleMeshPolyLine(vertex_coord,
                                                     boundary_id);

// Create the triangle mesh polygon for outer boundary
outer_boundary_pt = new TriangleMeshPolygon(outer_boundary_line_pt);
```

Next we define the curvilinear inner boundary in terms of two `TriangleMeshCurviLines` which define the hole in the domain:

```
// Inner circular boundary
//------------------------
Vector<TriangleMeshCurveSection*> inner_boundary_line_pt(2);

// The intrinsic coordinates for the beginning and end of the curve
double s_start = 0.0;
double s_end   = MathematicalConstants::Pi;
boundary_id = 0;
inner_boundary_line_pt[0]=
 new TriangleMeshCurviLine(inner_circle_pt,
                           s_start,
                           s_end,
                           n_segments,
                           boundary_id);

// The intrinsic coordinates for the beginning and end of the curve
s_start = MathematicalConstants::Pi;
s_end   = 2.0*MathematicalConstants::Pi;
boundary_id = 1;
inner_boundary_line_pt[1]=
 new TriangleMeshCurviLine(inner_circle_pt,
                           s_start,
                           s_end,
                           n_segments,
                           boundary_id);

// Combine to hole
//----------------
Vector<TriangleMeshClosedCurve*> hole_pt(1);
Vector<double> hole_coords(2);
hole_coords[0]=0.0;
hole_coords[1]=0.0;
hole_pt[0]=new TriangleMeshClosedCurve(inner_boundary_line_pt,
                                       hole_coords);
```

We specify the mesh parameters (including a target element size)

```
// Use the TriangleMeshParameters object for helping on the manage
// of the TriangleMesh parameters. The only parameter that needs to take
// is the outer boundary.
TriangleMeshParameters triangle_mesh_parameters(outer_boundary_pt);
// Specify the closed curve using the TriangleMeshParameters object
triangle_mesh_parameters.internal_closed_curve_pt() = hole_pt;
// Target element size in bulk mesh
triangle_mesh_parameters.element_area() = Global_Parameters::Element_area;
```

and build the bulk mesh

```
// Build the mesh
Solid_mesh_pt=new
 RefineableTriangleMesh<ELASTICITY_ELEMENT>(triangle_mesh_parameters);
```

We create the PML meshes and add them (and the solid mesh) to the Problem's collection of sub-meshes and build the global mesh.

```
// Create the main triangular mesh
add_sub_mesh(Solid_mesh_pt);
// Create PML meshes and add them to the global mesh
create_pml_meshes();
// Build the entire mesh from its submeshes
build_global_mesh();
```

Next we pass the problem parameters to all elements via calling `ElasticAnnulusProblem<ELASTICITY_ELEMENT>::complete_problem_setup()` discussed below (remember that even the elements in the PML layers need to be told about these since they adjust the γ_x

and γ_y functions in terms of these parameters), apply the boundary conditions and assign the equation numbers.

```
 // Complete problem setup
 complete_problem_setup();
 //Assign equation numbers
 cout << assign_eqn_numbers() << std::endl;
 // Set output directory
 Doc_info.set_directory(Global_Parameters::Directory);
} //end_of_constructor
```

The setup of the problem is now complete.

1.9.1 Completing the problem setup

We pass the problem parameters to all elements, which are used in the equation construction and perfectly matched layer definition.

```
//====================start_of_complete_problem_setup====================
/// Complete problem setup
//======================================================================
template<class ELASTICITY_ELEMENT>
void ElasticAnnulusProblem<ELASTICITY_ELEMENT>::complete_problem_setup()
{
#ifdef ADAPTIVE
 // Min element size allowed during adaptation
 if (!CommandLineArgs::command_line_flag_has_been_set("--validation"))
  {
   Solid_mesh_pt->min_element_size()=1.0e-5;
  }
#endif
 //Assign the physical properties to the elements
 //----------------------------------------------
 unsigned nel=this->mesh_pt()->nelement();
 for (unsigned e=0;e<nel;e++)
  {
   /// \short Upcast from PMLElement to time harmonic
   /// linear elasticity bulk element
   PMLTimeHarmonicLinearElasticityEquations<2> *el_pt =
    dynamic_cast<PMLTimeHarmonicLinearElasticityEquations<2>*>
    (mesh_pt()->element_pt(e));

   // Set the constitutive law
   el_pt->elasticity_tensor_pt() = Global_Parameters::E_pt;

   // Square of non-dim frequency
   el_pt->omega_sq_pt()= &Global_Parameters::Omega_sq;
  } // end_of_assignment
```

We pin all four nodal values (representing the real and imaginary part of the two displacement components) on the inner boundaries (boundaries 0 and 1; see enumeration of the boundaries in the constructor) and assign the desired boundary values.

```
 for(unsigned b=0;b<2;b++)
  {
   unsigned n_node = Solid_mesh_pt->nboundary_node(b);
   for (unsigned n=0;n<n_node;n++)
    {
       Node* nod_pt=Solid_mesh_pt->boundary_node_pt(b,n);
       Vector<double> x_node(2);
       x_node[0]=nod_pt->x(0);
       x_node[1]=nod_pt->x(1);
       Vector<double> u_exact(4);
       Global_Parameters::exact_u(x_node,u_exact);
       for (unsigned k=0;k<4;k++)
        {
         nod_pt->pin(k);
         nod_pt->set_value(k,u_exact[k]);
        }
    }
  }
} // end_of_complete_setup
```

1.10 Post-processing

As expected, the `doc_solution(...)` member function documents the computed solution. We are particularly interested in the computed solution and the exact solution for comparison purposes

```
 // Output displacement field
 //--------------------------
 sprintf(filename,"%s/soln_bulk%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Solid_mesh_pt->output(some_file,n_plot);
 some_file.close();
  // Output exact solution
 //----------------------
```

```
 sprintf(filename,"%s/exact_soln%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 Solid_mesh_pt->output_fct(some_file,n_plot,Global_Parameters::exact_u);
 some_file.close();
```

We also output the solution in the perfectly matched layers themselves and document the norm of the solution, the wall clock time and number of degrees of freedom.

```
 // Output runtime (wall clock time) in s in a file
 sprintf(filename,"%s/wall_clock_time%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 some_file << Global_Parameters::T_end-Global_Parameters::T_start << std::endl;
 some_file.close();
 // Output number of degrees of freedom in a file
 sprintf(filename,"%s/ndof%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 some_file << this->ndof() << std::endl;
 some_file.close();
 // Output norm of solution (to allow validation of solution even
 // if triangle generates a slightly different mesh)
 sprintf(filename,"%s/elast_soln_norm%i.dat",Doc_info.directory().c_str(),
         Doc_info.number());
 some_file.open(filename);
 double norm_soln=0.0;
 this->mesh_pt()->compute_norm(norm_soln);
 some_file << sqrt(norm_soln) << std::endl;
 cout << "Norm of computed solution: "   << sqrt(norm_soln)  << endl;
 // Increment label for output files
 Doc_info.number()++;
} //end doc
```

1.11 Comments and Exercises

1.11.1 Comments

- If you inspect the `driver code` you will notice that it also contains relevant code to perform spatially adaptive simulations of the problem – the adaptive version of the code is selected with `#ifdefs`.
- The choice for the absorbing functions in our implementation of the PMLs is not unique. There are alternatives varying in both order and continuity properties. The current form is the result of several feasibility studies and comparisons found in `Bermudez et al.` For Helmholtz equations these damping functions produce an acceptable result in most practical situations without further modifications. For very specific applications, alternatives may need to be used and can easily be implemented within the existing framework.

1.11.2 Exercises

1.11.2.1 Changing perfectly matched layer parameters

Confirm that only a relatively small number of PML elements (across the thickness of the PML layer) is required to effectively damp the outgoing waves. It is also interesting to explore the effect of increasing the number of elements of the mesh inside the perfectly matched layers.
Another parameter that can be adjusted is the geometrical thickness of the perfectly matched layers. In the case of linear elasticity (as opposed to for example the Helmholtz equation), the thickness of these layers must be relatively large to obtain the best results. Try to explain why and take note of the differences between the two mentioned problems.

1.11.2.2 Spatial adaptivity

The driver code discussed above already contains the straightforward modifications required to enable spatial adaptivity. Explore this (by recompiling the code with -DADAPTIVE) and explain why spatial adaptivity is not particularly helpful for the test problem discussed above.

1.11.2.3 Linear and cubic finite elements

The driver code also contains (commented out) modifications that allow the simulation to be performed with three-node (linear) and ten-node (cubic) triangles. Explore the performance of these elements and confirm that the helper functions correctly create matching (four-node and sixteen-node) quad elements in the PML layers.

1.11.2.4 Default values for problem parameters

Following our usual convention, we provide default values for problem parameters where this is sensible. Some parameters, such as the elasticity tensor, do need to be set since there are no obvious defaults. If `oomph-lib` is compiled in `PARANOID` mode, an error is thrown if the relevant pointers haven't been set. Without paranoia, you get a segmentation fault...
Confirm that this is the case by commenting out the relevant assignments.

1.11.2.5 Non-convex PML boundaries

As discussed above, we currently provide helper functions to attach PML layers to axis-aligned boundaries of 2D meshes with convex outer boundaries. Essentially, this restricts us to rectangular computational domains. Extend this capability by developing methodologies to

- deal with non-convex domain boundaries. We suggest you create PML meshes for the non-convex corners first, then create the axis-aligned meshes (note that these have to share nodes with the already-created elements that occupy the non-convex corners), and then create the corner meshes for the convex corners (as before). When you're done, let us know – this would be a really useful addition to oomph-lib's machinery. We're happy to help!
- Repeat the same exercise in 3D – somewhat less trivial (so we're even keener for somebody to have a go!)

1.12 Source files for this tutorial

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

  demo_drivers/pml_time_harmonic_linear_elasticity/

- The driver code is:

  demo_drivers/pml_time_harmonic_linear_elasticity/time_harmonic_↩
  elasticity_driver.cc

1.13 PDF file

A pdf version of this document is available.