Chapter 1

Example problem: The Helmholtz equation with perfectly matched layers

In this document we discuss the finite-element-based solution of the Helmholtz equation with the Summerfeld boundary condition, an elliptic PDE that describes time-harmonic wave propagation problems. Compared to the "standard form" of the Helmholtz equation, discussed in another tutorial, the formulation used here allows the imposition of the Sommerfeld radiation condition by means of so-called "perfectly matched layers" (PMLs) as an alternative to classical absorbing/approximate boundary conditions or DtN maps.

We start by reviewing the relevant theory and then present the solution of a simple model problem – the outward propagation of waves from the surface of a cylinder.

Acknowledgement This tutorial and the associated driver codes were developed jointly with Radu Cimpeanu (Imperial College London)

1.1 Theory: The Helmholtz equation for time-harmonic scattering problems

The Helmholtz equation governs time-harmonic solutions of problems governed by the linear wave equation

$$\nabla(c^2\nabla U(x,y,t)) = \frac{\partial^2 U(x,y,t)}{\partial t^2}, \qquad (1)$$

where c is the wavespeed. Assuming that $U(x,y,t)$ is time-harmonic, with frequency ω, we write the real function $U(x,y,t)$ as

$$U(x,y,t) = Re(u(x,y)\, e^{-i\omega t})$$

where $u(x,y)$ is complex-valued. If the wavespeed is constant this transforms (1) into the standard form of the Helmholtz equation

$$\nabla^2 u(x,y) + k^2 u(x,y) = 0 \qquad (2)$$

where

$$k = \frac{\omega}{c} \qquad (3)$$

is the wave number. Like other elliptic PDEs the Helmholtz equation admits Dirichlet, Neumann (flux) and Robin boundary conditions.

If the equation is solved in an infinite domain (e.g. in scattering problems) the solution must satisfy the so-called Sommerfeld radiation condition which in 2D has the form

$$\lim_{r\to\infty} \sqrt{r}\left(\frac{\partial u}{\partial r} - iku\right) = 0.$$

Mathematically, this condition is required to ensure the uniqueness of the solution (and hence the well-posedness of the problem). In a physical context, such as a scattering problem, the condition ensures that scattering of an incoming wave only produces outgoing not incoming waves from infinity.

1.2 Discretisation by finite elements

We provide separate storage for the real and imaginary parts of the solution – each Node therefore stores two unknowns values. By default, the real and imaginary parts are stored as values 0 and 1, respectively; see the section The enumeration of the unknowns for details.

The application of Dirichlet and Neumann boundary conditions is straightforward:

- Dirichlet conditions are imposed by pinning the relevant nodal values and setting them to the appropriate prescribed values.
- Neumann (flux) boundary conditions are imposed via FaceElements (here the PMLHelmholtzFlux↩ Elements). As usual we attach these to the faces of the "bulk" elements that are subject to the Neumann boundary conditions.

The imposition of the Sommerfeld radiation condition for problems in infinite domains is slightly more complicated. Another tutorial shows how to impose this condition by means of absorbing/approximate boundary conditions or DtN maps. In the next section we will discuss an alternative approach to this problem by means of perfectly matched layers.

1.2.1 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 the Sommerfeld radiation condition ensuring the suitable decay of the solution at large distances from the region of interest (the vicinity of the scatterer, say).

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.

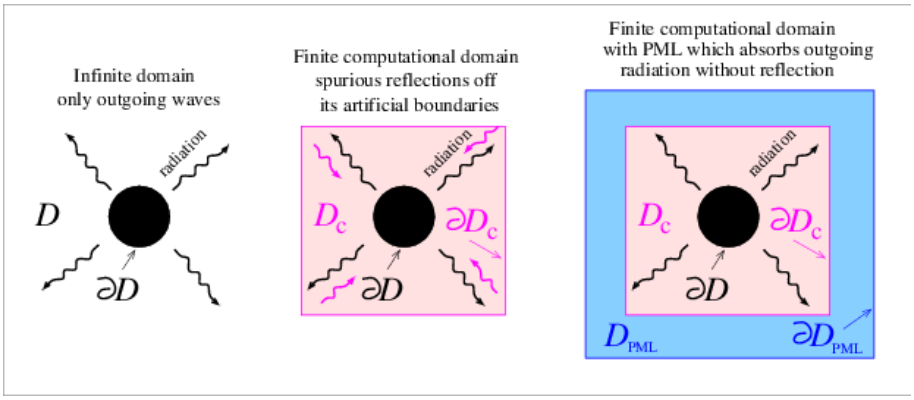

Figure 1.1 Sketch illustrating the idea behind perfectly matched layers.

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 further generalisation of the equations, achieved by introducing the complex coordinate mapping

$$\frac{\partial}{\partial x_j} \rightarrow \frac{1}{\gamma_j}\frac{\partial}{\partial x_j} \qquad (4)$$

within the perfectly matched layers. This makes the problem anisotropic and in 2D we have

$$\nabla^2 u = \frac{1}{\gamma_x}\frac{\partial}{\partial x}\left(\frac{1}{\gamma_x}\frac{\partial u}{\partial x}\right) + \frac{1}{\gamma_y}\frac{\partial}{\partial y}\left(\frac{1}{\gamma_y}\frac{\partial u}{\partial y}\right) \qquad (5)$$

The choice of γ_x and γ_y 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:

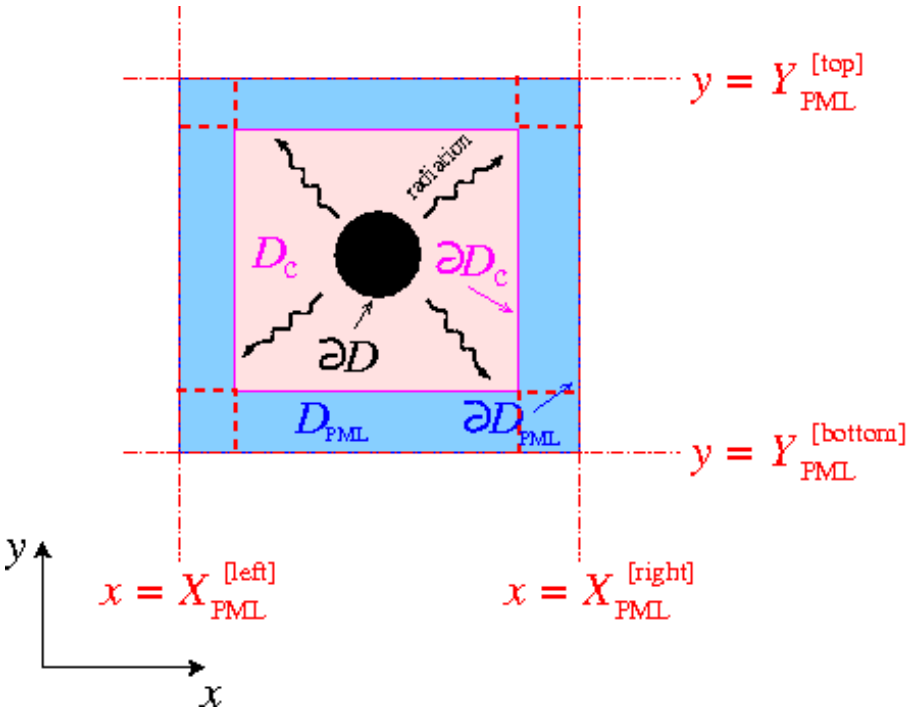

Figure 1.2 Sketch illustrating the geometry of the perfectly matched layers.

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

$$\gamma_x(x) = 1 + \frac{i}{k}\,\sigma_x(x) \qquad \text{with} \quad \sigma_x(x) = \frac{1}{|X_{PML} - x|}, \qquad (6)$$

  where X_{PML} 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) we set

$$\gamma_x = 1,$$

  and

$$\gamma_y(y) = 1 + \frac{i}{k}\,\sigma_y(y) \qquad \text{with} \quad \sigma_y(y) = \frac{1}{|Y_{PML} - y|}, \qquad (7)$$

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

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

$$\gamma_x(x) = 1 + \frac{i}{k}\,\sigma_x(x) \qquad \text{with} \quad \sigma_x(x) = \frac{1}{|X_{PML} - x|} \tag{8}$$

and

$$\gamma_y(y) = 1 + \frac{i}{k}\,\sigma_y(y) \qquad \text{with} \quad \sigma_y(y) = \frac{1}{|Y_{PML} - y|}. \tag{9}$$

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

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

1.2.2 Implementation of the perfectly matched layers within oomph-lib

The finite-element-discretised equations (2) (modified by the PML terms discussed above) are implemented in the `PMLHelmholtzEquations<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 `QPMLHelmholtzElement<2,NNODE_1D>` elements and the parameter `NNODE_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

```
PMLHelper::create_right_pml_mesh(bulk_mesh_pt, b_bulk, n_pml, width);
```

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 A specific example: Outward propagation of acoustic waves from the surface of a cylindrical object

We will now demonstrate the methodology for a specific example: the propagation of axisymmetric waves from the surface of a circular disk. This is a good test case because any deviations from the axisymmetry of the (exact) solution by spurious reflections from the boundaries of the computational domain are easy to detect visually.

The specific domain used in this case can be seen in the figure below. We create an unstructured mesh of six-noded `TPMLHelmholtzElements` to create the finite computational domain surrounding a circular disk. This is surrounded by four axis-aligned PML layers and four corner meshes (each made of nine-noded `QPMLHelmholtz↩Elements`).

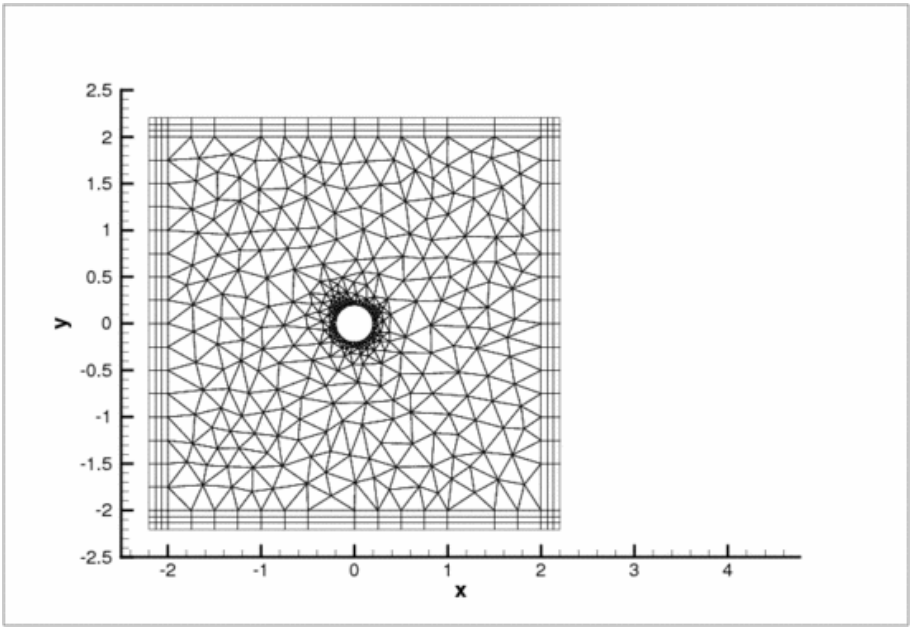

Figure 1.3 The computational domain used in the example problem.

1.4 Results

The figures below show the real part of the solution $Re(u(x,y,t))$ radiating from a circular disk with a radius of $r = 0.1$ for the case when the non-zero Dirichlet boundary conditions are imposed only on the real part of u by setting

$$Re(u(x,y))\Big|_{\partial D} = 0.1 \quad and \quad Im(u(x,y))\Big|_{\partial D} = 0.$$

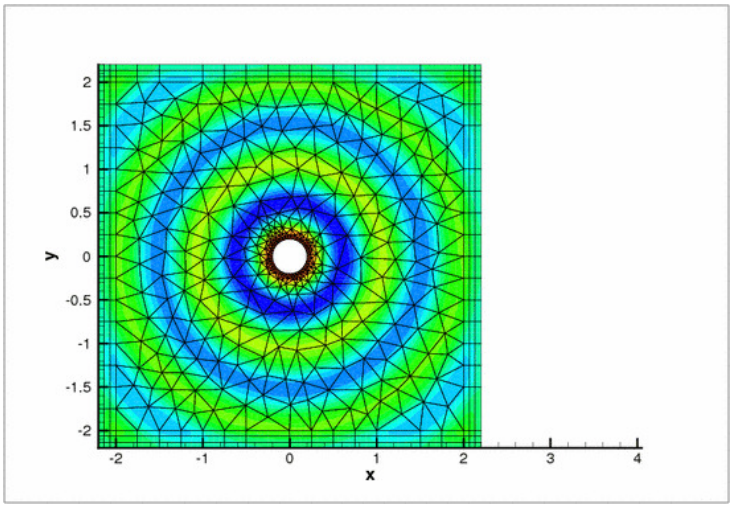

Figure 1.4 Sample solution with activated perfectly matched layers.

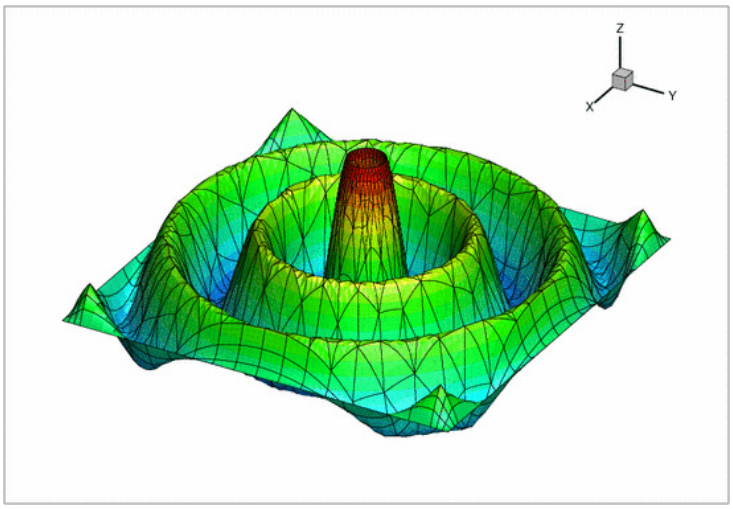

Figure 1.5 Sample solution with activated perfectly matched layers -- height view.

From the two images, one can notice the clean circular solution across the domain. Had the perfectly matched layers not been effective, numerical artifacts would have been observed throughout the domain.

This is demonstrated by the following two figures which show the solution obtained without the PML layers (and "do-nothing" (zero-flux) boundary conditions on the outer boundaries of the computational domain). The spurious reflections from the boundaries completely dominate the solution which bears no resemblance to the exact solution.

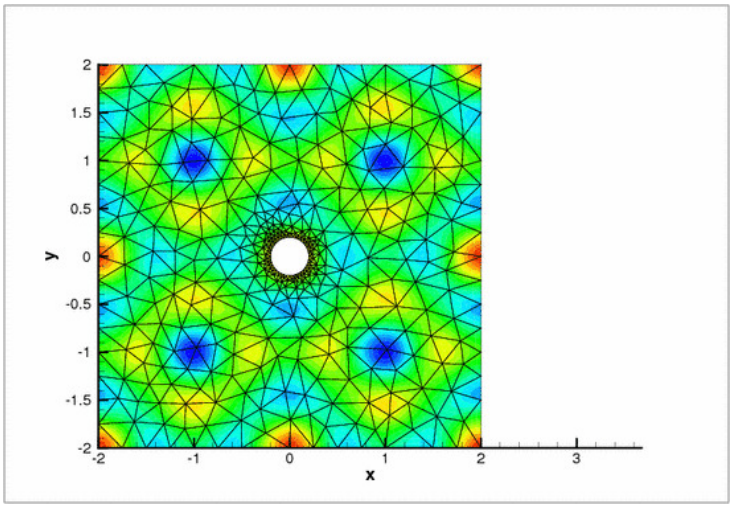

Figure 1.6 The solution with zero flux boundary conditions.

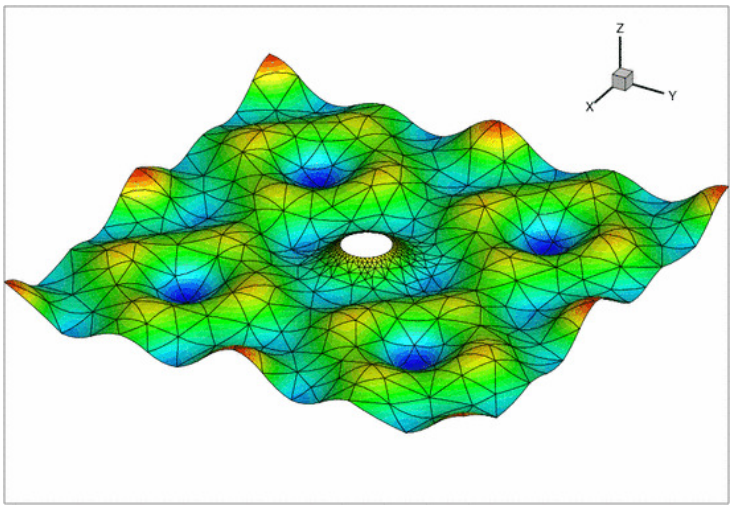

Figure 1.7 The solution with zero flux boundary conditions -- height view.

1.5 The numerical solution

1.5.1 The global namespace

As usual, we define the problem parameters in a global namespace. After non-dimensionalisation, the only parameter is wavenumber, k.

```
//===== start_of_namespace=============================================
/// Namespace for the Helmholtz problem parameters
//=====================================================================
namespace GlobalParameters
{

 /// Wavenumber (also known as k), k=omega/c
 double Wavenumber = sqrt(50.0);

 /// Square of the wavenumber (also known as k^2)
 double K_squared = Wavenumber * Wavenumber;

} // end of namespace
```

1.5.2 The driver code

The driver code is very straightforward. We start by building the `Problem` object, using six-noded triangular generalised Helmholtz elements:

```
//==========start_of_main==============================================
/// Solve 2D Helmholtz problem
//=====================================================================
int main(int argc, char **argv)
{
 // Set up the problem with 2D six-node elements from the
 // TPMLHelmholtzElement family.
 PMLProblem<TPMLHelmholtzElement<2,3> >  problem;
```

Next we define the output directory.

```
 // Create label for output
 //------------------------
 DocInfo doc_info;

 // Set output directory
 doc_info.set_directory("RESLT");
```

Finally, we solve the problem and document the results.

```
 // Solve the problem with Newton's method
 problem.newton_solve();
 //Output solution
 problem.doc_solution(doc_info);

} //end of main
```

1.5.3 The problem class

The problem class is very similar to that employed for the `solution of the 2D Helmholtz equation with flux 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).

```
//========= start_of_problem_class====================================
/// Problem class to demonstrate use of perfectly matched layers
/// for Helmholtz problems.
//====================================================================
template<class ELEMENT>
class PMLProblem : public Problem
{

public:

 /// Constructor
 PMLProblem();

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

 /// Update the problem specs before solve (empty)
 void actions_before_newton_solve(){}

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

 /// Doc the solution. DocInfo object stores flags/labels for where the
 /// output gets written to
 void doc_solution(DocInfo& doc_info);

 /// Create PML meshes
 void create_pml_meshes();

 // Apply boundary conditions
 void apply_boundary_conditions();
```

The private member data includes pointers the bulk mesh

```
 /// Pointer to the "bulk" mesh
 TriangleMesh<ELEMENT>* Bulk_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;

 /// Trace file
 ofstream Trace_file;

}; // end of problem class
```

1.5.4 The problem constructor

We start by creating the `Circle` object that defines the inner boundary of the domain.

```
//=======start_of_constructor=============================================
/// Constructor for Helmholtz problem
```

```
//========================================================================
template<class ELEMENT>
PMLProblem<ELEMENT>::PMLProblem()
{

 // Open trace file
 Trace_file.open("RESLT/trace.dat");

 // 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 = 20;
 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() = 0.1;
```

and build the bulk mesh

```
// Build "bulk" mesh
Bulk_mesh_pt=new TriangleMesh<ELEMENT>(triangle_mesh_parameters);
```

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

```
// Create the main triangular mesh
add_sub_mesh(Bulk_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 (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:

```
// Let's have a look where the boundaries are
this->mesh_pt()->output("global_mesh.dat");
this->mesh_pt()->output_boundaries("global_mesh_boundary.dat");

// Complete the build of all elements so they are fully functional
unsigned n_element = this->mesh_pt()->nelement();
for(unsigned e=0;e<n_element;e++)
 {
  // Upcast from GeneralisedElement to Helmholtz bulk element
  PMLHelmholtzEquations<2> *el_pt =
   dynamic_cast<PMLHelmholtzEquations<2>*>(mesh_pt()->element_pt(e));

  //Set the k_squared double pointer
  el_pt->k_squared_pt() = &GlobalParameters::K_squared;
 }

// Apply boundary conditions
```

```
 apply_boundary_conditions();

 // Setup equation numbering scheme
 cout <<"Number of equations: " << assign_eqn_numbers() << std::endl;

} // end of constructor
```

The problem is now ready to be solved.

1.5.5 Applying the boundary conditions

We pin both nodal values (representing the real and imaginary part of the solutions) on the inner boundaries (boundaries 0 and 1; see enumeration of the boundaries in the constructor) and assign the desired boundary values.

```
//==================start_of_apply_boundary_conditions====================
/// Apply boundary conditions
//========================================================================
template<class ELEMENT>
void PMLProblem<ELEMENT>::apply_boundary_conditions()
{

 // Boundary conditions are set on the surface of the circle
 // as a constant nonzero Dirichlet boundary condition
 unsigned n_bound = Bulk_mesh_pt->nboundary();

 for(unsigned b=0;b<n_bound;b++)
  {
   unsigned n_node = Bulk_mesh_pt->nboundary_node(b);
   for (unsigned n=0;n<n_node;n++)
    {
     if ((b==0) || (b==1))
      {
       Node* nod_pt=Bulk_mesh_pt->boundary_node_pt(b,n);
       nod_pt->pin(0);
       nod_pt->pin(1);

       nod_pt->set_value(0,0.1);
       nod_pt->set_value(1,0.0);
      }
    }
  }

}// end of apply_boundary_conditions
```

1.5.6 Post-processing

The post-processing function `doc_solution(...)` simply outputs the computed solution.

```
//======================start_of_doc======================================
/// Doc the solution: doc_info contains labels/output directory etc.
//========================================================================
template<class ELEMENT>
void PMLProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{

 ofstream some_file,some_file2;
 char filename[100];

 // Number of plot points
 unsigned npts;
 npts=5;

 // Output solution
 //-----------------
 snprintf(filename, sizeof(filename), "%s/soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Bulk_mesh_pt->output(some_file,npts);
 some_file.close();
```

1.6 Comments and Exercises

1.6.1 The enumeration of the unknowns

As discussed in the introduction, most practically relevant solutions of the Helmholtz equation are complex valued. Since `oomph-lib's` solvers only deal with real (double precision) unknowns, the equations are separated into their real and imaginary parts. In the implementation of the Helmholtz elements, we store the real and imaginary parts of the solution as two separate values at each node. By default, the real and imaginary parts are accessible via `Node::value(0)` and `Node::value(1)`. However, to facilitate the use of the elements in multi-physics problems we avoid accessing the unknowns directly in this manner but provide the virtual function
`std::complex<unsigned> HelmholtzEquations<DIM>::u_index_helmholtz()`

which returns a complex number made of the two unsigneds that indicate which nodal value represents the real and imaginary parts of the solution. This function may be overloaded in combined multi-physics elements in which a Helmholtz element is combined (by multiple inheritance) with another element, using the strategy described in the Boussinesq convection tutorial.

1.6.2 PML damping functions

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 both Bermudez et al. and in the relevant papers on Radu Cimpeanu's webpage. 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.

1.6.3 Exercises

1.6.3.1 Changing perfectly matched layer parameters

Confirm that only a very small number of PML elements (across the thickness of the PML layer) is required to effectively damp the outgoing waves. Furthermore, show that (and try to explain why) PMLs with too many elements may not perform as expected.

A second parameter that can be adjusted is the geometrical thickness of the perfectly matched layers. Relative thin layers layer tend to perform better than thick layers with few elements across their width. Confirm this and try to find an explanation for the phenomenon, given the form of the absorbing functions used in the complex coordinate transformation.

1.6.3.2 Large wavenumbers

For Helmholtz problems in general, ill-conditioning appears as the wavenumber becomes very large. By altering wavespeed and/or frequency, explore the limitations of both the mesh and the solver in terms of this parameter. Try adjusting the target element size in order to alleviate resolution-related effects. Assess the effectiveness of the perfectly matched layers in high wavenumber problems.

1.6.3.3 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.

1.6.3.4 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.

1.6.3.5 Default values for problem parameters

Following our usual convention, we provide default values for problem parameters where this is sensible. For instance, the PML mapping function defaults to the one proposed by Bermudez et al. as this appears to be optimal. Some parameters, such as the wavenumber squared k^2 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.

1.6.3.6 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!)

1.7 Source files for this tutorial

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

  `demo_drivers/pml_helmholtz/scattering/`

- The driver code is:

  `demo_drivers/pml_helmholtz/scattering/unstructured_two_d_helmholtz.cc`

1.8 PDF file

A `pdf version` of this document is available. \