Chapter 1

Demo problem: How to create simple refineable meshes

In a previous example we gave an overview of oomph-lib's powerful mesh adaptation capabilities and demonstrated the use of the functions

- Problem::refine_uniformly() which performs automatic, uniform refinement of a given (refineable) mesh.
- Problem::adapt() which performs automatic mesh adaptation (local refinement or unrefinement), based on error estimates that are computed (automatically) by a chosen error estimator.
- Problem::newton_solve(...) – a black-box adaptive Newton solver that automatically adapts the mesh and recomputes the solution until it satisfies the prescribed error bounds.

Provided the problem has been discretised with suitable "refineable mesh" and "refineable element" objects from oomph-lib's mesh and finite element libraries, none of these functions require any intervention by the user. Most of oomph-lib finite elements are already available in "refineable" and "non-refineable" forms. For instance, the RefineableQPoissonElement that we used in the previous example is the refineable equivalent of the 2D QPoissonElement. Another document describes how to create new refineable elements. Here we shall discuss how to "upgrade" existing meshes to RefineableMeshes, i.e. meshes that can be used with oomph-lib's mesh adaptation routines.

The minimum functionality that must be provided by such meshes is specified by the pure virtual functions in the abstract base class RefineableMesh and all refineable Meshes should be derived from this class. Here is a graphical representation of the typical inheritance structure for refineable meshes, illustrated for 2D quad meshes:

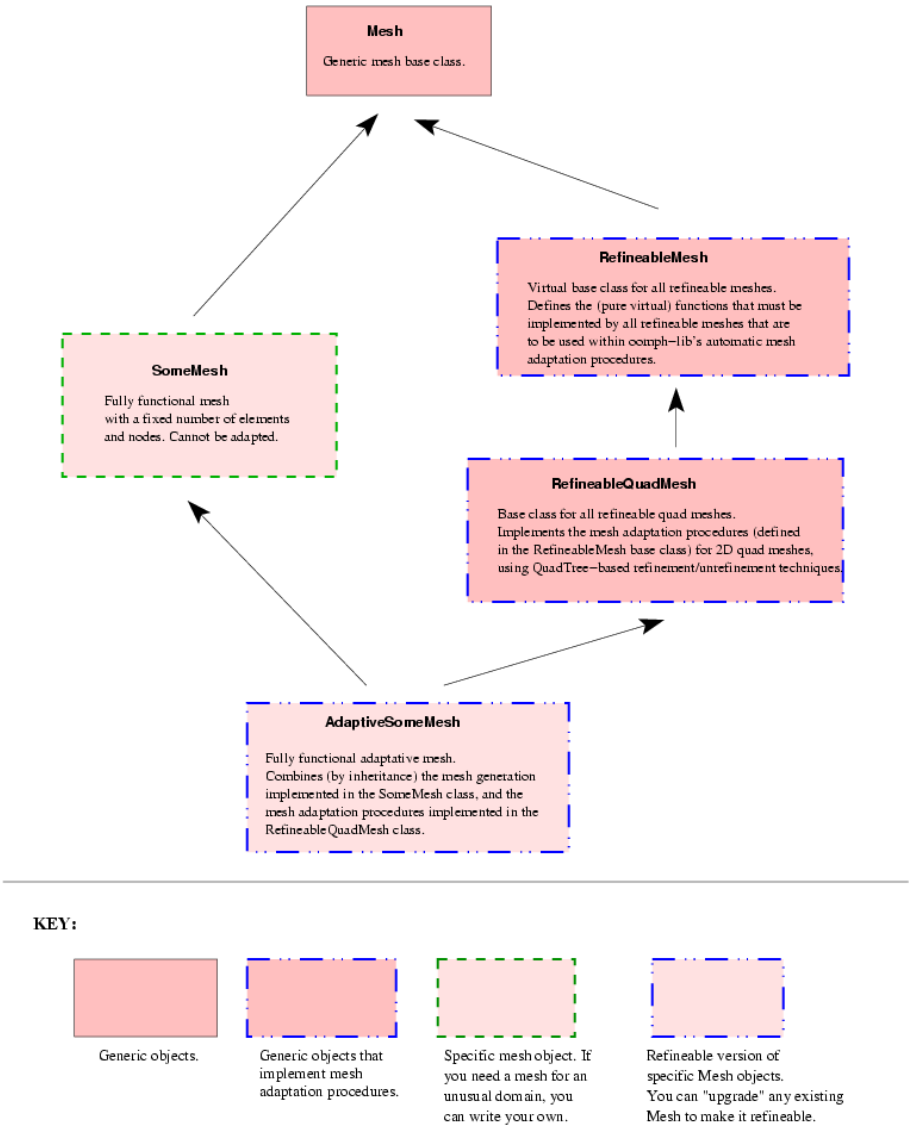

Figure 1.1 Typical inheritance structure for refineable meshes, illustrated for 2D quad meshes.

The diagram contains two fully-functional meshes:

- The `SomeMesh` is some basic, non-refineable mesh that is derived directly from the generic `Mesh` base class. Typically, it provides a coarse discretisation of a 2D domain with 2D elements from the `QElement` family. Its constructor creates the mesh's nodes and elements and initialises the various boundary lookup schemes. (Consult the `"How to build a mesh"` section of the `Quick Guide` for details of the generic mesh generation process.)
- The `RefineableSomeMesh` is the refineable equivalent of the basic `SomeMesh`. It inherits the original mesh layout from the `SomeMesh` class. Refineability is added by inheriting from the `RefineableQuad↩Mesh` class; this class implements the mesh adaptation procedures, specified as pure virtual functions in the `RefineableMesh` class, for 2D quad meshes, employing `QuadTree` - based refinement techniques.

Equivalent inheritance structures can be/are implemented for meshes with different element topologies: For instance, the `RefineableBrickMesh` class is the 3D equivalent of the `RefineableQuadMesh` class: It performs the mesh adaptation for 3D brick meshes by `OcTree` - based refinement techniques.

Typically, most of the "hard work" involved in the mesh adaptation process is implemented in the intermediate classes (such as `RefineableQuadMesh` or `RefineableBrickMesh`). Upgrading an existing mesh to a refineable version therefore usually requires very little effort. We demonstrate this by re-visiting the 2D Poisson problem that we analysed in an `earlier example:`

Two-dimensional model Poisson problem

Solve

$$\sum_{i=1}^{2} \frac{\partial^2 u}{\partial x_i^2} = f(x_1, x_2), \qquad (1)$$

in the rectangular domain $D = \{(x_1, x_2) \in [0, 1] \times [0, 2]\}$, with Dirichlet boundary conditions

$$u|_{\partial D} = u_0 \qquad (2)$$

where

$$u_0(x_1, x_2) = \tanh(1 - \alpha(x_1 \tan \Phi - x_2)) \qquad (3)$$

and

$$f(x_1, x_2) = \sum_{i=1}^{2} \frac{\partial^2 u_0}{\partial x_i^2} \qquad (4)$$

so that $u_0(x_1, x_2)$ represents the exact solution of the problem.

Recall that for large values of α the solution approaches a step function

$$u_{step}(x_1, x_2) = \begin{cases} -1 & \text{for } x_2 < x_1 \tan \Phi \\ 1 & \text{for } x_2 > x_1 \tan \Phi \end{cases}$$

Accurate numerical solution can therefore only be obtained if the mesh is refined – ideally only in the vicinity of the "step":

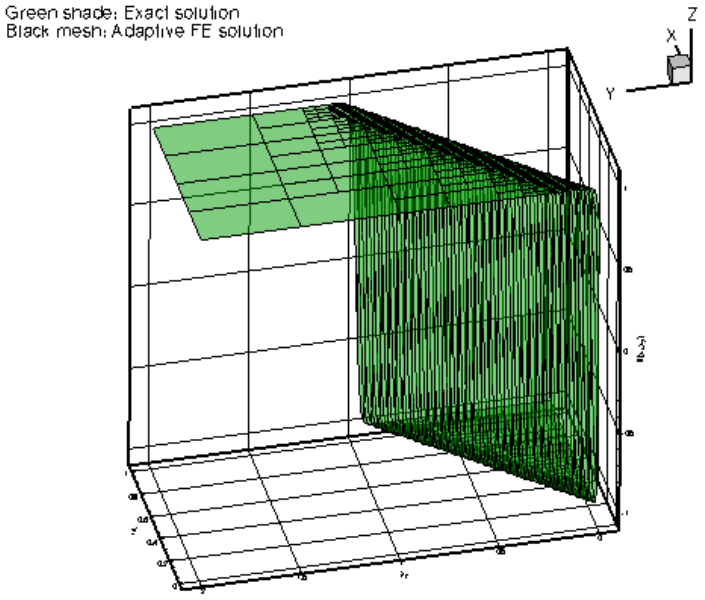

Figure 1.2 Plot of the solution with adaptive mesh refinement

We shall discuss the driver code
two_d_poisson_adapt.cc which solves the above problem with adaptive mesh refinement. Its key feature

is the creation of the refineable mesh SimpleRefineableRectangularQuadMesh – the refineable equivalent of the SimpleRectangularQuadMesh used in the earlier example.

1.1 Creating the refineable mesh

QuadTree-based mesh refinement, as implemented in the RefineableQuadMesh class, requires the coarse initial mesh to be represented by a QuadTreeForest: Each element in the mesh must be associated with a QuadTree, and the relative orientation of the various QuadTrees relative to each other must be established. This can be done automatically by calling the function RefineableQuadMesh::setup_quadtree_forest(). The SimpleRefineableRectangularQuadMesh class is therefore very compact. The mesh is derived from the SimpleRectangularQuadMesh and the Refineable1QuadMesh classes, both of which are templated by the element type:

```
//=============================start_of_mesh======================
/// Refineable equivalent of the SimpleRectangularQuadMesh.
/// Refinement is performed by the QuadTree-based procedures
/// implemented in the RefineableQuadMesh base class.
//================================================================
template<class ELEMENT>
class SimpleRefineableRectangularQuadMesh :
 public virtual SimpleRectangularQuadMesh<ELEMENT>,
 public RefineableQuadMesh<ELEMENT>
```

The mesh constructor first calls the constructor of the underlying SimpleRectangularQuadMesh to create the nodes and elements, and to set up the various boundary lookup schemes. The call to RefineableQuadMesh::setup_quadtree_forest() creates the QuadTreeForest representation of the mesh. That's all!

```
public:

 ///  Pass number of elements in the horizontal
 /// and vertical directions, and the corresponding dimensions.
 /// Timestepper defaults to Static.
 SimpleRefineableRectangularQuadMesh(const unsigned &Nx,
                                     const unsigned &Ny,
                                     const double &Lx, const double &Ly,
                                     TimeStepper* time_stepper_pt=
                                     &Mesh::Default_TimeStepper) :
  SimpleRectangularQuadMesh<ELEMENT>(Nx,Ny,Lx,Ly,time_stepper_pt)
  {
   // Nodal positions etc. were created in constructor for
   // SimpleRectangularQuadMesh<...> --> We only need to set up
   // adaptivity information: Associate finite elements with their
   // QuadTrees and plant them in a QuadTreeForest:
   this->setup_quadtree_forest();
  } // end of constructor
```

The destructor can remain empty, as all memory de-allocation is handled in the mesh base classes.

```
 /// Destructor: Empty
 virtual ~SimpleRefineableRectangularQuadMesh() {}
```

1.2 Global parameters and functions

The specification of the source function and the exact solution in the namespace TanhSolnForPoisson is identical to that in the non-refineable version discussed in the previous example.

1.3 The driver code

The driver code is very similar to that in the non-refineable version. We simply change the mesh from the SimpleRectangularQuadMesh to its refineable equivalent, and discretise the problem with nine-node RefineableQPoissonElements instead of nine-node 2D QPoissonElements. We choose a large value of $\alpha = 50$ for the "steepness" parameter and solve the problem with the "black-box" Newton solver, allowing for up to four adaptive refinements:

```
//===== start_of_main=========================================================
/// Driver code for 2D Poisson problem
//============================================================================
int main()
{
 //Set up the problem
 //------------------
 // Create the problem with 2D nine-node refineable elements from the
 // RefineableQuadPoissonElement family. Pass pointer to source function.
 RefineablePoissonProblem<RefineableQPoissonElement<2,3> >
```

```
  problem(&TanhSolnForPoisson::get_source);

 // Create label for output
 //------------------------
 DocInfo doc_info;
 // Set output directory
 doc_info.set_directory("RESLT");
 // Step number
 doc_info.number()=0;
 // Check if 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);
  }

 // Set the orientation of the "step" to 45 degrees
 TanhSolnForPoisson::TanPhi=1.0;

 // Choose a large value for the steepness of the "step"
 TanhSolnForPoisson::Alpha=50.0;
 // Solve the problem, performing up to 4 adaptive refinements
 problem.newton_solve(4);

 //Output the solution
 problem.doc_solution(doc_info);

} //end of main
```

1.4 The problem class

The problem class definition is virtually identical to that in the non-refineable version. The only new function is an overloaded version of the `Problem::mesh_pt()` function which returns a pointer to the generic `Mesh` object. Our version returns a pointer to the specific mesh, to avoid the use of explicit casts in the rest of the code.

```
//====== start_of_problem_class=====================================
/// 2D Poisson problem on rectangular domain, discretised with
/// refineable 2D QPoisson elements. The specific type of element is
/// specified via the template parameter.
//==================================================================
template<class ELEMENT>
class RefineablePoissonProblem : public Problem
{
public:

 /// Constructor: Pass pointer to source function
 RefineablePoissonProblem(PoissonEquations<2>::PoissonSourceFctPt
                          source_fct_pt);

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

 /// Update the problem specs before solve: Reset boundary conditions
 /// to the values from the exact solution.
 void actions_before_newton_solve();

 /// Update the problem 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);

 /// Overloaded version of the Problem's access function to
 /// the mesh. Recasts the pointer to the base Mesh object to
 /// the actual mesh type.
 SimpleRefineableRectangularQuadMesh<ELEMENT>* mesh_pt()
  {
   return dynamic_cast<SimpleRefineableRectangularQuadMesh<ELEMENT>*>(
    Problem::mesh_pt());
  }
private:

 /// Pointer to source function
 PoissonEquations<2>::PoissonSourceFctPt Source_fct_pt;
}; // end of problem class
```

[See the discussion of the 1D Poisson problem for a more detailed discussion of the function type PoissonEquations<2>::PoissonSourceFctPt.]

1.5 The Problem constructor

The problem constructor is virtually identical to that in the non-refineable version. The only change required is the specification of an error estimator for the mesh adaptations: We create an instance of the Z2ErrorEstimator and pass a pointer to it to the mesh.

```
//=====start_of_constructor=============================================
/// Constructor for Poisson problem: Pass pointer to source function.
//======================================================================
template<class ELEMENT>
RefineablePoissonProblem<ELEMENT>::
      RefineablePoissonProblem(PoissonEquations<2>::PoissonSourceFctPt
                               source_fct_pt)
       :  Source_fct_pt(source_fct_pt)
{
 // Setup mesh
 // # of elements in x-direction
 unsigned n_x=4;
 // # of elements in y-direction
 unsigned n_y=4;
 // Domain length in x-direction
 double l_x=1.0;
 // Domain length in y-direction
 double l_y=2.0;
 // Build and assign mesh
 Problem::mesh_pt() =
  new SimpleRefineableRectangularQuadMesh<ELEMENT>(n_x,n_y,l_x,l_y);
 // Create/set error estimator
 mesh_pt()->spatial_error_estimator_pt()=new Z2ErrorEstimator;

 // Set the boundary conditions for this problem: All nodes are
 // free by default -- only need to pin the ones that have Dirichlet conditions
 // here.
 unsigned num_bound = mesh_pt()->nboundary();
 for(unsigned ibound=0;ibound<num_bound;ibound++)
  {
   unsigned num_nod= mesh_pt()->nboundary_node(ibound);
   for (unsigned inod=0;inod<num_nod;inod++)
    {
     mesh_pt()->boundary_node_pt(ibound,inod)->pin(0);
    }
  }
 // Complete the build of all elements so they are fully functional
 // Loop over the elements to set up element-specific
 // things that cannot be handled by the (argument-free!) ELEMENT
 // constructor: Pass pointer to source function
 unsigned n_element = mesh_pt()->nelement();
 for(unsigned i=0;i<n_element;i++)
  {
   // Upcast from GeneralsedElement to the present element
   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));
   //Set the source function pointer
   el_pt->source_fct_pt() = Source_fct_pt;
  }
 // Setup equation numbering scheme
 cout <<"Number of equations: " << assign_eqn_numbers() << std::endl;
} // end of constructor
```

1.6 "Actions before solve"

This function is identical to that in the non-refineable version.

1.7 Post-processing

The function doc_solution(...) is identical to that in the non-refineable version.

1.8 Comments

Since most of the "hard work" involved in the mesh adaptation is hidden from "user" we briefly comment on various steps involved in the mesh adaptation process and highlight certain important implications.

1.8.1 The refinement pattern

The QuadTree - based mesh adaption routines, implemented in the RefineableQuadMesh class, split elements into four "son elements" if the error estimate exceeds the acceptable maximum. By default, the position of any newly created nodes is determined from the geometric mapping of the "father" element. For instance, when a four-node quad "father" element is split into four "sons", five new nodes are created and they are located at $(s_0, s_1) = (0, -1),\ (1, 0),\ (0, 1),\ (-1, 0)$ and $(0, 0)$ in the father element's local coordinate system. This procedure is adequate for problems in which the coarse initial mesh provides a perfect representation of the domain (e.g. polygonal domains). If the domain has curvilinear boundaries, successive mesh refinements must generate a more and more accurate representation of these boundaries. This requires slight changes to the mesh adaptation procedures. We will discuss these in another example.
The splitting of "father" elements into four equal-sized "sons" maintains the aspect ratio of the elements during the mesh adaptation. The good news is that mesh adaption will not cause a deterioration in the element quality. The bad news is that poorly designed coarse meshes cannot be improved by mesh adaptation. It is therefore worthwhile to invest some time into the initial mesh design. For complicated domains, it may be sensible to perform the initial mesh generation with a dedicated, third-party mesh generator. (We provide another example to illustrate how to build oomph-lib meshes based on the output from a third-party mesh generator.)

1.8.2 Hanging nodes

The local splitting of elements can create so-called "hanging nodes" – nodes on element edges that are not shared by any adjacent elements. The nodal values and coordinates at such nodes must be constrained to ensure the inter-element continuity of the solution. Specifically, the nodal values and coordinates at hanging nodes must be suitable linear combinations of the values at a number of "master nodes". (In the first instance, the master nodes are the nodes on the adjacent element's edge that *are* shared by adjacent elements. If there are multiple levels of refinement, such nodes can themselves be hanging; the ultimate set of master nodes is therefore be determined recursively.)
The setup of the hanging node constraints is handled automatically by the mesh adaptation routines and the technical details are therefore of little relevance to the general user. (The "bottom up" discussion of the data structure provides details if you are interested.) One aspect of the way in which hanging nodes are handled in oomph-lib is important, however. Up to now we have accessed nodal values either via the function
`Node::set_value(...)`
which sets the values stored at a Node, or the pointer-based access function
`Node::value_pt(...)`
which returns a pointer to these values.
What happens when a node is hanging, i.e. if Node::is_hanging() returns true?

A convention

The functions
`Node::set_value(...)`
and
`Node::value_pt(...)`
always refer to the nodal values stored at the `Node` itself.
Important: If a node is hanging, the value pointed to by `Node::value_pt(...)` is **not** kept up to date!
The correctly constrained nodal value must be computed "on the fly", using the list of master nodes and their respective weights, stored in the node's `HangingInfo` object. This is done automatically by the function
`Node::value(...)`
which returns the appropriate value for hanging *and* non-hanging nodes: For non-hanging nodes it returns the value pointed to by `Node::value_pt(...)`; for hanging nodes, it computes the correctly constrained values. When developing new elements or writing new post-processing routines, the user should therefore always refer to nodal values with the `Node::value(...)` function to ensure that the code works correctly in the presence of hanging nodes.
We provide equivalent functions to access the nodal positions: The function
`Node::x(...)`
returns the values of (Eulerian) coordinates stored at the node. These values can be out of date if the node is hanging. The function
`Node::position(...)`
should be used to determine a node's Eulerian position – this function is the equivalent of `Node::value(...)` and determines the nodal coordinates of hanging nodes "on the fly", using the node's list of master nodes and weights.
Finally, we note that while the nodal values and coordinates stored at a node might be out of date *while* a node is hanging, the values are automatically assigned up-to-date values when subsequent mesh adaptations change a node's status from hanging to non-hanging.

1.9 Source files for this tutorial

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

  demo_drivers/poisson/two_d_poisson_adapt/

- The driver code is:

  demo_drivers/poisson/two_d_poisson_adapt/two_d_poisson_adapt.cc

1.10 PDF file

A pdf version of this document is available.