Chapter 1

Demo problem: Adaptive solution of Poisson's equation in a fish-shaped domain.

In this document, we discuss the solution of a 2D Poisson problem using `oomph-lib's` powerful mesh adaptation routines:

Two-dimensional model Poisson problem in a non-trivial domain

Solve

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

in the fish-shaped domain D_{fish}, with homogeneous Dirichlet boundary conditions

$$u|_{\partial D_{fish}} = 0. \qquad (2)$$

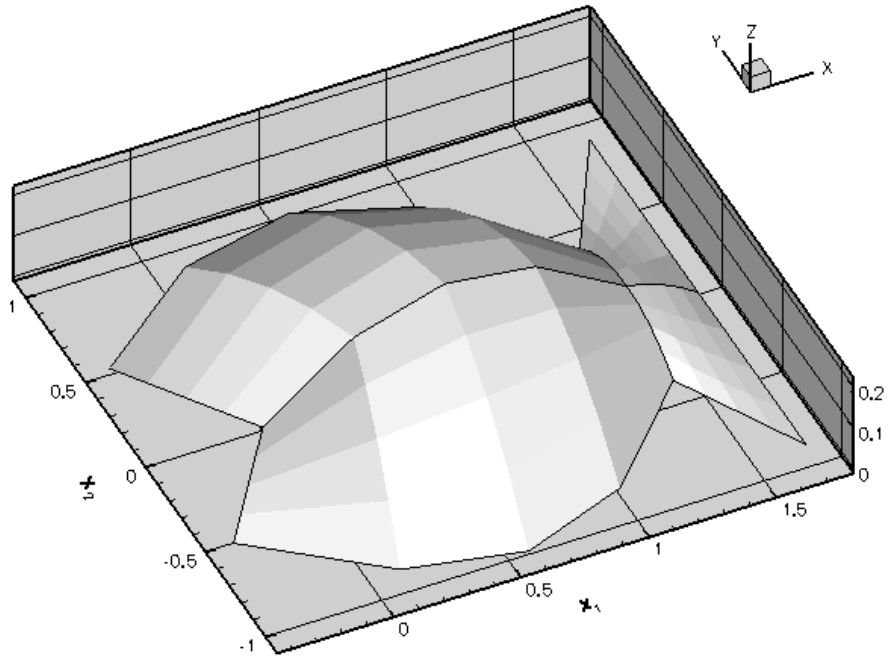

Figure 1.1 Plot of the solution

The sharp corners in the domain create singularities in the solution (its derivatives are unbounded) and so accurate results can only be obtained if we use a sufficiently fine discretisation. Implementing this by uniform mesh refinement would create a huge number of elements in the interior of the domain where the fine discretisation is not required.

To avoid this problem, `oomph-lib` provides mesh adaptation routines that automatically adapt the mesh, based on *a posteriori* error estimates. Regions in which an error estimator indicates that the solution is not resolved to the required accuracy are refined; automatic unrefinement is performed in regions where the discretisation is unnecessarily fine.

We provide a detailed discussion of the driver code `fish_poisson.cc` which illustrates a variety of mesh refinement procedures. [The alternative driver code `fish_poisson_no_adapt.cc` solves the same problem without mesh adaptation. Its structure is very similar to that in the `2D Poisson problem considered earlier`. It is provided mainly to illustrate how easy to it is incorporate adaptivity into a `Problem`.]

In the current example we demonstrate how to *use* existing refineable meshes and elements. Two further examples will demonstrate how easy it is to *create* refineable meshes in `domains with polygonal boundaries` and in `domains with curvilinear boundaries.`

1.1 Global parameters and functions

The namespace `ConstSourceForPoisson` only contains the constant source function $f(x) = -1$.

```
//============ start_of_namespace=====================================
/// Namespace for const source term in Poisson equation
//====================================================================
namespace ConstSourceForPoisson
{

 /// Strength of source function: default value -1.0
 double Strength=-1.0;

/// Const source function
 void source_function(const Vector<double>& x, double& source)
 {
  source = Strength;
 }

} // end of namespace
```

1.2 The driver code

The main code is very short and calls two functions that illustrate two different adaptation strategies:

- A black-box approach in which the adaptation cycle

  1. solve the problem on the initial, coarse mesh
  2. compute an error estimate
  3. adapt the mesh
  4. solve again

  is performed automatically until the solution satisfies the required error bounds (or until the maximum permitted number of adaptation steps has been reached).

- In the second approach we start by performing a number of uniform mesh refinement steps, and then use incremental adaptations, allowing us to document how the refinement proceeds.

```
//================start_of_main=============================================
/// Demonstrate how to solve 2D Poisson problem in
/// fish-shaped domain with mesh adaptation.
//============================================================================
int main()
{
 // Solve with adaptation, docing the intermediate steps
 solve_with_incremental_adaptation();

 // Solve directly, with fully automatic adaptation
 solve_with_fully_automatic_adaptation();

} // end of main
```

1.2.1 Black-box adaptation

We start by creating the Problem object, using the refineable equivalent of the `QPoissonElement` – the `RefineableQPoissonElement`, which is templated by the dimension and the number of nodes along the element's edges; the `RefineableQPoissonElement<2,3>` is a nine-node (bi-quadratic) quad element.

```
//==============================start_black_box===========================
/// Demonstrate how to solve 2D Poisson problem in
/// fish-shaped domain with fully automatic mesh adaptation
//========================================================================
void solve_with_fully_automatic_adaptation()
{

  //Set up the problem with nine-node refineable Poisson elements
  RefineableFishPoissonProblem<RefineableQPoissonElement<2,3> > problem;
```

After creating the `DocInfo` object, we document the (default) adaptivity targets:

```
  // Setup labels for output
  //------------------------
  DocInfo doc_info;

  // Set output directory
  doc_info.set_directory("RESLT_fully_automatic");

  // Step number
  doc_info.number()=0;


  // Doc (default) refinement targets
  //---------------------------------
  problem.mesh_pt()->doc_adaptivity_targets(cout);
```

These include

- **The target for the maximum error:** Any elements whose error estimate exceed this value will be split into four "sons".
- **The target for the minimum error:** Any elements whose error estimate lies below this value are deemed to be unnecessarily small and are scheduled for (possible) unrefinement. [Elements can only be unrefined (i.e. merged with their "brothers") if their "brothers" are also scheduled for unrefinement.]
- **The minimum refinement level:** In problems with curvilinear domain boundaries it is often necessary to retain a reasonably accurate representation of the boundary (e.g. for postprocessing purposes), even if the error estimate suggests that the mesh could be unrefined further.

- **The maximum refinement level:** In problems where the solution has singularities, the refinement process would continue indefinitely, therefore an upper bound on the refinement level must be imposed.
- Finally, because unrefinement is done purely to speed up the computation, it would not make sense to adapt the mesh if this process would only remove a few elements, while forcing the re-computation of the solution on an only slightly coarsened mesh. Therefore, no mesh adaptation is performed if
  - the adaptation would only perform unrefinements
  - **and** the number of elements scheduled for unrefinement is below a certain threshold.

These default parameters can be changed by the user; see Comments and Exercises.
The fully-adaptive solution of the problem is very simple. We simply pass the maximum number of adaptations to the Newton solver and document the results. Done!

```
  // Solve/doc the problem with fully automatic adaptation
  //------------------------------------------------------

  // Maximum number of adaptations:
  unsigned max_adapt=5;

  // Solve the problem; perform up to specified number of adaptations.
  problem.newton_solve(max_adapt);

  //Output solution
  problem.doc_solution(doc_info);

} // end black box
```

1.2.2 Incremental adaptation

To allow the user more control over the mesh adaptation process, `oomph-lib` provides a number of functions that perform individual adaptation steps without re-computing the solution immediately. This allows the user to

- perform uniform mesh refinement and unrefinement,
- impose a specific refinement pattern,
- monitor/document the progress of the automatic adaptation.

The second driver function illustrates some of these functions. We start by setting up the problem, create the `DocInfo` object and document the adaptivity targets, exactly as before:

```
//======================start_of_incremental==============================
/// Demonstrate how to solve 2D Poisson problem in
/// fish-shaped domain with mesh adaptation. First we solve on the original
/// coarse mesh. Next we do a few uniform refinement steps and re-solve.
/// Finally, we enter into an automatic adapation loop.
//========================================================================
void solve_with_incremental_adaptation()
{
 //Set up the problem with nine-node refineable Poisson elements
 RefineableFishPoissonProblem<RefineableQPoissonElement<2,3> > problem;
 // Setup labels for output
 //------------------------
 DocInfo doc_info;
 // Set output directory
 doc_info.set_directory("RESLT_incremental");
 // Step number
 doc_info.number()=0;


 // Doc (default) refinement targets
 //---------------------------------
 problem.mesh_pt()->doc_adaptivity_targets(cout);
```

Next, we solve the problem on the original, very coarse mesh and document the result:

```
 // Solve/doc the problem on the initial, very coarse mesh
 //-------------------------------------------------------
 // Solve the problem
 problem.newton_solve();
 //Output solution
 problem.doc_solution(doc_info);
 //Increment counter for solutions
 doc_info.number()++;
```

We know that the result is unlikely to be very accurate, so we apply three levels of uniform refinement, increasing the number of elements from 4 to 256, and re-compute:

```
 // Do three rounds of uniform mesh refinement and re-solve
 //--------------------------------------------------------
 problem.refine_uniformly();
 problem.refine_uniformly();
```

```
 problem.refine_uniformly();
 // Solve the problem
 problem.newton_solve();
 //Output solution
 problem.doc_solution(doc_info);
 //Increment counter for solutions
 doc_info.number()++;
```

The solution looks much smoother but we suspect that the corner regions are still under-resolved. Therefore, we call the `Problem::adapt()` function which computes an error estimate for all elements and automatically performs a single mesh adaptation (refinement/unrefinement) step. If this adaptation changes the mesh, we recompute the solution, using the "normal" Newton solver without automatic adaptation. We document the solution and continue the adaptation cycle until `Problem::adapt()` ceases to change the mesh:

```
 // Now do (up to) four rounds of fully automatic adapation in response to
 //-------------------------------------------------------------------------
 // error estimate
 //---------------
 unsigned max_solve=4;
 for (unsigned isolve=0;isolve<max_solve;isolve++)
  {
   // Adapt problem/mesh
   problem.adapt();

   // Re-solve the problem if the adaptation has changed anything
   if ((problem.mesh_pt()->nrefined()  !=0)||
       (problem.mesh_pt()->nunrefined()!=0))
    {
     problem.newton_solve();
    }
   else
    {
     cout « "Mesh wasn't adapted --> we'll stop here" « std::endl;
     break;
    }

   //Output solution
   problem.doc_solution(doc_info);

   //Increment counter for solutions
   doc_info.number()++;
  }
} // end of incremental
```

The progress of the adaptation is illustrated in the animated gif at the beginning of this document. The first frame displays the solution on the original four-element mesh; the next frame shows the solution on the uniformly refined mesh; the final two frames show the progress of the subsequent, error-estimate-driven mesh adaptation.

1.3 The problem class

The problem class is virtually identical to that used in the 2D Poisson problem without mesh refinement. In the present problem, we leave the function `Problem::actions_before_newton_solve()` empty because the boundary conditions do not change. The function `RefineableFishPoissonProblem::mesh_p` overloads the (virtual) function `Problem::mesh_pt()` since it returns a pointer to a generic `Mesh` object, rather than a pointer to the specific mesh used in this problem. This avoids explicit re-casts in the rest of the code where member functions of the specific mesh need to be accessed.

```
//======start_of_problem_class=========================================
/// Refineable Poisson problem in fish-shaped domain.
/// Template parameter identifies the element type.
//======================================================================
template<class ELEMENT>
class RefineableFishPoissonProblem : public Problem
{

public:

 /// Constructor
 RefineableFishPoissonProblem();

 /// Destructor: Empty
 virtual ~RefineableFishPoissonProblem(){}

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

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

 /// Overloaded version of the problem's access function to
 /// the mesh. Recasts the pointer to the base Mesh object to
 /// the actual mesh type.
```

```
 RefineableFishMesh<ELEMENT>* mesh_pt()
  {
   return dynamic_cast<RefineableFishMesh<ELEMENT>*>(Problem::mesh_pt());
  }

 /// Doc the solution. Output directory and labels are specified
 /// by DocInfo object
 void doc_solution(DocInfo& doc_info);

}; // end of problem class
```

1.4 The Problem constructor

We start by creating the mesh, using `oomph-lib's RefineableFishMesh` object:

```
//==========start_of_constructor=========================================
/// Constructor for adaptive Poisson problem in fish-shaped
/// domain.
//========================================================================
template<class ELEMENT>
RefineableFishPoissonProblem<ELEMENT>::RefineableFishPoissonProblem()
{

 // Build fish mesh -- this is a coarse base mesh consisting
 // of four elements. We'll refine/adapt the mesh later.
 Problem::mesh_pt()=new RefineableFishMesh<ELEMENT>;
```

Next, we create an error estimator for the problem. The `Z2ErrorEstimator` is based on Zhu and Zienkiewicz's flux recovery technique and can be used with all elements that are derived from the `ElementWithZ2Error↩ Estimator` base class (or with functions that implement the pure virtual functions that are defined in this class) – the `RefineableQPoissonElement` is an element of this type.

```
 // Create/set error estimator
 mesh_pt()->spatial_error_estimator_pt()=new Z2ErrorEstimator;
```

Next we pin the nodal values on all boundaries, apply the homogeneous Dirichlet boundary conditions, pass the pointer to the source function to the elements, and set up the equation numbering scheme.

```
 // Set the boundary conditions for this problem: All nodes are
 // free by default -- just pin the ones that have Dirichlet conditions
 // here. Since the boundary values are never changed, we set
 // them here rather than in actions_before_newton_solve().
 unsigned n_bound = mesh_pt()->nboundary();
 for(unsigned i=0;i<n_bound;i++)
  {
   unsigned n_node = mesh_pt()->nboundary_node(i);
   for (unsigned n=0;n<n_node;n++)
    {
     // Pin the single scalar value at this node
     mesh_pt()->boundary_node_pt(i,n)->pin(0);

     // Assign the homogenous boundary condition for the one and only
     // nodal value
     mesh_pt()->boundary_node_pt(i,n)->set_value(0,0.0);
    }
  }

 // Loop over elements and set pointers to source function
 unsigned n_element = mesh_pt()->nelement();
 for(unsigned e=0;e<n_element;e++)
  {
   // Upcast from FiniteElement to the present element
   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e));

   //Set the source function pointer
   el_pt->source_fct_pt() = &ConstSourceForPoisson::source_function;
  }

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

} // end of constructor
```

1.5 Post-processing

The post-processing routine writes the computed result to an output file, labeled with the identifiers specified in the `DocInfo` object.

```
//=======start_of_doc=====================================================
/// Doc the solution in tecplot format.
//========================================================================
template<class ELEMENT>
void RefineableFishPoissonProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
```

```
{

 ofstream some_file;
 char filename[100];

 // Number of plot points in each coordinate direction.
 unsigned npts;
 npts=5;

 // Output solution
 sprintf(filename,"%s/soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 mesh_pt()->output(some_file,npts);
 some_file.close();
} // end of doc
```

1.6 Comments and Exercises

The purpose of this example was to provide a high-level overview of `oomph-lib's` mesh adaptation procedures. We demonstrated that the implementation of full adaptivity only required us to

- replace the `FishMesh` and the `QPoissonElement` objects by their refineable equivalents, `RefineableFishMesh` and `RefineableQPoissonElement`, respectively
- specify the error estimator, and
- specify the maximum number of adaptations for the black-box adaptive Newton solver.

(Compare the Problem specification for the current problem to that of its non-refineable equivalent, contained in the alternative driver code `fish_poisson_no_adapt.cc`.)
Since most of the "hard work" involved in the mesh adaptation is "hidden" from the user, we highlight some important aspects of the procedure:

1.6.1 Automatic transfer of the solution/boundary conditions during the mesh adaptation

The `Problem::adapt()` function automatically determines the correct boundary conditions for newly created nodes on the Mesh boundary; it automatically updates the equation numbering scheme, and interpolates the solution from the original mesh onto the adapted mesh. This is important in nonlinear problems where the provision of a good initial guess for the Newton iteration is vital; and in time-dependent problems where the solution at one timestep provides initial conditions for the next one. See the discussion of `the adaptive solution of the unsteady heat equation` for more details. Furthermore, the source function pointers are automatically passed to an element's four "son" elements when the element is subdivided. This allows the adaptation to proceed completely automatically, without any intervention by the "user". On return from `Problem::adapt()` the problem can immediately be re-solved.
In some special cases, certain actions may need to be performed before or after the mesh adaptation (e.g. if flux boundary conditions are applied by `FaceElements`; this is explained in `another example`). To ensure that these steps are performed when the adaptation is controlled by the "black-box" adaptive Newton solver, the `Problem` class provides the two empty virtual functions
`Problem::actions_before_adapt()`
and
`Problem::actions_after_adapt()`
which are called automatically before and after the adaptation. The "user" can overload these in his/her specific `Problem` class to implement such actions.

1.6.2 Automatic mesh adaptation in domains with curvilinear boundaries

The mesh adaptation not only increases the number of elements but also produces a more accurate representation of the curvilinear domain boundary – new boundary nodes are placed exactly onto the analytically-defined, curvilinear boundary, rather than on the boundaries of the "father" element, which only provides an approximate representation of the exact domain boundary. This is achieved by employing a `MacroElement-based` representation of the `Domain` – we will discuss this in more detail in `another example.`

1.6.3 Problem adaptation vs. Mesh adaptation

Many adaptation routines in the `Problem` class have equivalents in the `RefineableMesh` class. It is important to appreciate the important differences between them: If adaptation is performed at the `Problem` level, the adapted `Problem` is fully functional, i.e. boundary conditions will have been assigned for newly created nodes on the mesh boundary, the equation numbering scheme will have been updated, etc. The adapted `Problem` can therefore be re-solved immediately. Conversely, if a mesh is refined directly, using the member functions of the `Refineable↩ Mesh` class, many of these additional tasks need to be performed "by hand" before the adapted `Problem` can be resolved.

1.6.4 Exercises

To familiarise yourself with `oomph-lib's` mesh adaptation procedures we suggest the following exercises:

1. When the Poisson problem is solved with the default refinement targets, no elements are unrefined. Increase the minimum permitted error from its default value of 10^{-5} to 10^{-4} by adding the statement
   ```
   problem.mesh_pt()->min_permitted_error()=1.0e-4;
   ```
   before
   ```
   problem.mesh_pt()->doc_adaptivity_targets(cout);
   ```
   This value forces an unrefinement of several elements in the mesh:

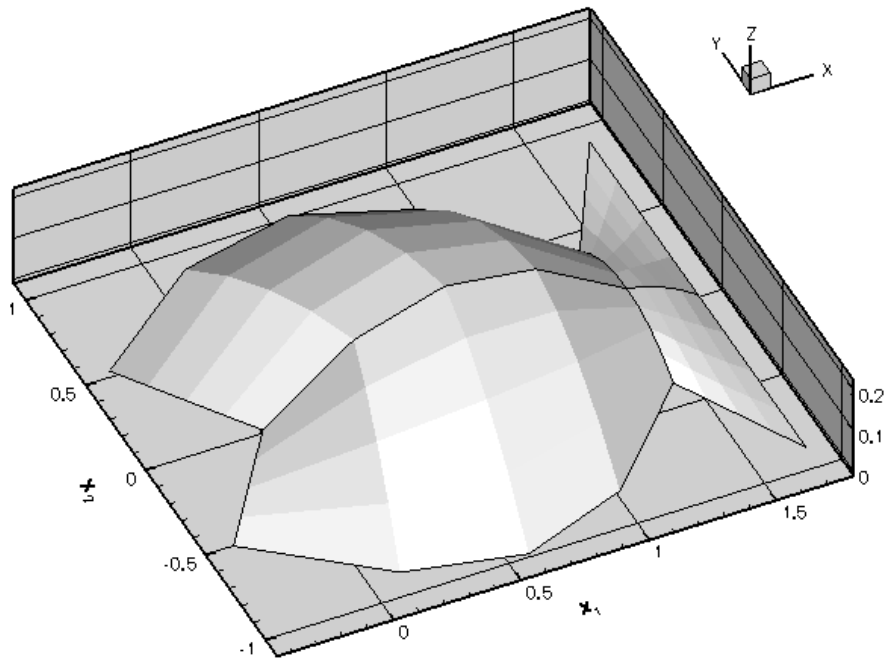

Figure 1.2 Plot of the solutions obtained with the modified adaptivity targets.

2. Convince yourself that `Problem::adapt()` does indeed interpolate the solution from the coarse mesh to the fine mesh – call `Problem::doc_solution(...)` before and after its execution.

3. The `Problem::refine_uniformly()` function has a counterpart `Problem::unrefine_↩ uniformly()`. Why does this function not simply unrefine every single element in the mesh? Explore the action of `Problem::unrefine_uniformly()` by plotting the solution before and after a few executions of this function.

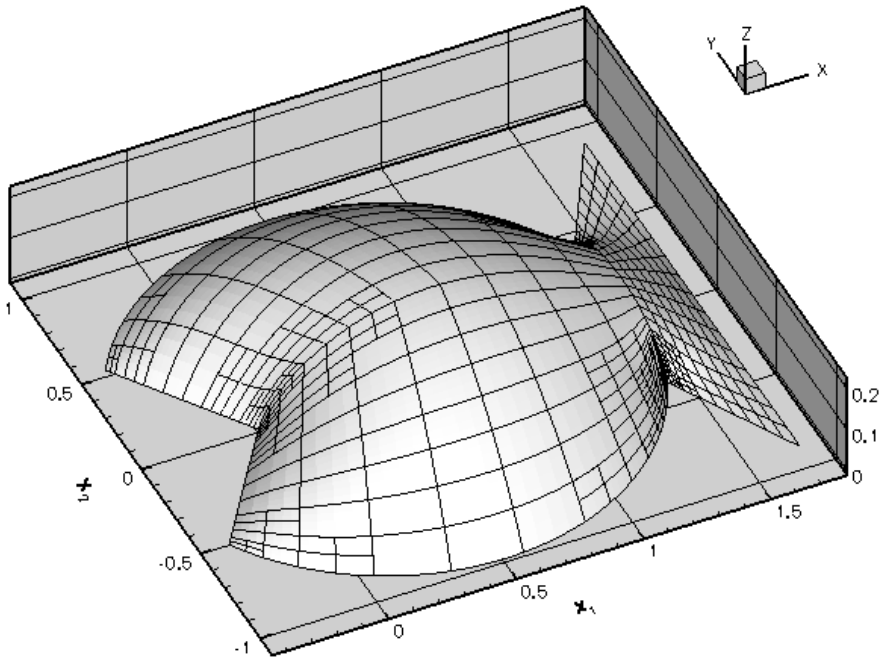

Figure 1.3 Uniform unrefinement

4. Impose a "user-defined" refinement pattern by calling the function `Problem::refine_selected_elements(...)`.

1.7 Source files for this tutorial

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

  demo_drivers/poisson/fish_poisson/

- The driver code is:

  demo_drivers/poisson/fish_poisson/fish_poisson.cc

1.8 PDF file

A pdf version of this document is available.