Chapter 1

Example problem: Adaptive solution of the 2D Poisson equation with flux boundary conditions

In this document we discuss the adaptive solution of a 2D Poisson problem with Neumann boundary conditions.

Two-dimensional model Poisson problem with Neumann boundary conditions

Solve

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

in the rectangular domain $D = \{(x_1, x_2) \in [0,1] \times [0,2]\}$. The domain boundary $\partial D = \partial D_{Neumann} \cup \partial D_{Dirichlet}$, where $\partial D_{Neumann} = \{(x_1, x_2) | x_1 = 1,\ x_2 \in [0,2]\}$. On $\partial D_{Dirichlet}$ we apply the Dirichlet boundary conditions

$$u|_{\partial D_{Dirichlet}} = u_0, \qquad (2)$$

where the function u_0 is given. On $\partial D_{Neumann}$ we apply the Neumann conditions

$$\left.\frac{\partial u}{\partial n}\right|_{\partial D_{Neumann}} = \left.\frac{\partial u}{\partial x_1}\right|_{\partial D_{Neumann}} = g_0, \qquad (3)$$

where the function g_0 is given.

In two previous examples we demonstrated two approaches for the *non-adaptive* solution of this problem. In both cases we created a "bulk" mesh of `QPoissonElements` and applied the flux boundary conditions by "attaching" `PoissonFluxElements` to the appropriate boundaries of the "bulk" Poisson elements. In the `first implementation` we simply added the pointers to the flux elements to the bulk mesh; in the `second implementation` we stored the surface and bulk elements in separate meshes and combined them to a single global mesh. We will now demonstrate that the second approach greatly facilitates the automatic problem adaptation. We use the mesh adaptation procedures of the `RefineableQuadMesh` class to adapt the bulk mesh, driven by the spatial error estimates in that mesh. The flux elements are neither involved in nor adapted by the refinement process of the bulk mesh. We therefore use the function `Problem::actions_before_adapt()` to

delete the flux elements before the adaptation, and `Problem::actions_after_adapt()` to re-attach them once the bulk mesh has been adapted.

As in the previous example we choose a source function and boundary conditions for which the function

$$u_0(x_1, x_2) = \tanh(1 - \alpha(x_1 \tan\Phi - x_2)), \qquad (4)$$

is the exact solution of the problem.

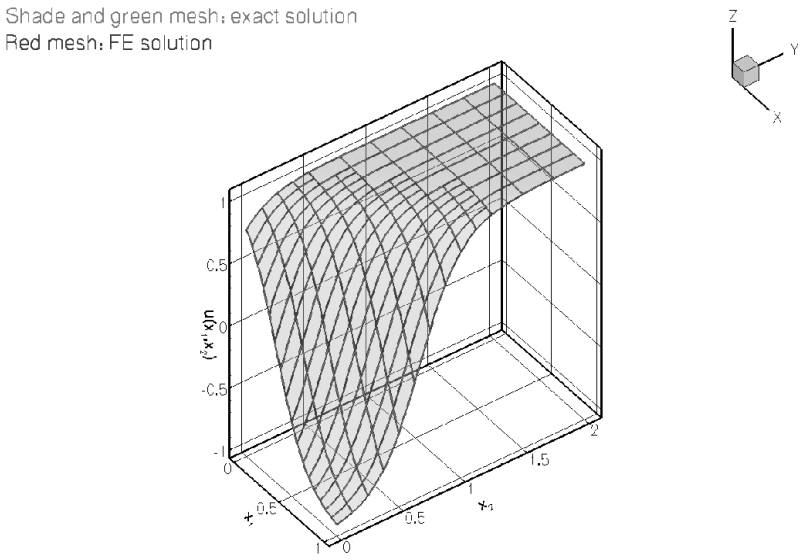

Figure 1.1 Plot of the solution obtained with automatic mesh adaptation

Since many functions in the driver code are identical to that in the non-adaptive version, discussed in the previous example, we only list those functions that differ. Please consult the source code two_d_poisson_flux_bc_adapt.cc for full details of the implementation.

1.1 Global parameters and functions

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

1.2 The driver code

The main code is virtually identical to that in the previous non-adaptive example . The only change is the provision of an argument to the Newton solver

```
// Solve the problem
problem.newton_solve(3);
```

which indicates that the problem should be adapted up to three times.

1.3 The problem class

The problem class is very similar to that in the non-adaptive implementation: The only difference is the provision of the functions `actions_before_adapt()`, `actions_after_adapt()`, `set_prescribed_flux_pt()`, and `delete_flux_elements(...)` which we discuss in more detail below.

```
//========= start_of_problem_class=====================================
/// 2D Poisson problem on rectangular domain, discretised with
/// 2D QPoisson elements. Flux boundary conditions are applied
/// along boundary 1 (the boundary where x=L). The specific type of
```

```
/// element is specified via the template parameter.
//======================================================================
template<class ELEMENT>
class RefineableTwoMeshFluxPoissonProblem : public Problem
{
public:

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

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

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

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

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

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

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

 /// \short Create Poisson flux elements on boundary b of the Mesh pointed
 /// to by bulk_mesh_pt and add them to the Mesh object pointed to by
 /// surface_mesh_pt
 void create_flux_elements(const unsigned &b, Mesh* const &bulk_mesh_pt,
                           Mesh* const &surface_mesh_pt);

 /// \short Delete Poisson flux elements and wipe the surface mesh
 void delete_flux_elements(Mesh* const &surface_mesh_pt);

 /// \short Set pointer to prescribed-flux function for all
 /// elements in the surface mesh
 void set_prescribed_flux_pt();

 /// Pointer to the "bulk" mesh
 SimpleRefineableRectangularQuadMesh<ELEMENT>* Bulk_mesh_pt;

 /// Pointer to the "surface" mesh
 Mesh* Surface_mesh_pt;

 /// 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.]

1.4 The Problem constructor

We create the bulk mesh and surface mesh as before. Next we create the spatial error estimator and pass it to the bulk mesh.
Apart from this, the problem is constructed as in the non-adaptive previous example.

1.5 Actions before adaptation

The mesh adaptation is driven by the error estimates in the bulk elements and only performed for that mesh. The flux elements must therefore be removed before adaptation. We do this by calling the function `delete_flux_elements(...)`, and then rebuilding the `Problem's` global mesh.

```
//====================start_of_actions_before_adapt=====================
/// Actions before adapt: Wipe the mesh of prescribed flux elements
//======================================================================
template<class ELEMENT>
void RefineableTwoMeshFluxPoissonProblem<ELEMENT>::actions_before_adapt()
{
 // Kill the flux elements and wipe surface mesh
 delete_flux_elements(Surface_mesh_pt);

 // Rebuild the Problem's global mesh from its various sub-meshes
 rebuild_global_mesh();
}// end of actions_before_adapt
```

1.6 Actions after adapt

After the (bulk-)mesh has been adapted, the flux elements must be re-attached. This is done by calling the function `create_flux_elements(...)`, followed by a rebuild of the `Problem's` global mesh. Finally, we set the function pointer to the prescribed flux function for each flux element.

```
//======================start_of_actions_after_adapt=======================
///  Actions after adapt: Rebuild the mesh of prescribed flux elements
//=========================================================================
template<class ELEMENT>
void RefineableTwoMeshFluxPoissonProblem<ELEMENT>::actions_after_adapt()
{
 // Create prescribed-flux elements from all elements that are
 // adjacent to boundary 1 and add them to surfac mesh
 create_flux_elements(1,Bulk_mesh_pt,Surface_mesh_pt);

 // Rebuild the Problem's global mesh from its various sub-meshes
 rebuild_global_mesh();

 // Set pointer to prescribed flux function for flux elements
 set_prescribed_flux_pt();

 // Doc refinement levels in bulk mesh
 unsigned min_refinement_level;
 unsigned max_refinement_level;
 Bulk_mesh_pt->get_refinement_levels(min_refinement_level,
                                     max_refinement_level);
 cout << "Min/max. refinement levels in bulk mesh: "
      << min_refinement_level << " "
      << max_refinement_level << std::endl;

}// end of actions_after_adapt
```

1.7 Delete flux elements

This function loops over all the flux elements (i.e. those in the surface mesh) and deletes them and their storage.

```
//============start_of_delete_flux_elements==============================
/// Delete Poisson Flux Elements and wipe the surface mesh
//========================================================================
template<class ELEMENT>
void RefineableTwoMeshFluxPoissonProblem<ELEMENT>::
delete_flux_elements(Mesh* const &surface_mesh_pt)
{
 // How many surface elements are in the surface mesh
 unsigned n_element = surface_mesh_pt->nelement();
 // Loop over the surface elements
 for(unsigned e=0;e<n_element;e++)
  {
   // Kill surface element
   delete surface_mesh_pt->element_pt(e);
  }
 // Wipe the mesh
 surface_mesh_pt->flush_element_and_node_storage();
} // end of delete_flux_elements
```

IMPORTANT: Note how the elements are first deleted "manually" and then "flushed" from the surface mesh, using the function `Mesh::flush_element_and_node_storage()`. This is necessary because deleting the surface mesh directly (by `delete surface_mesh_pt;`) would also delete its constituent `Nodes` which are shared with the bulk mesh and must therefore be retained!

1.8 Actions before solve

This remains as `before.`

1.9 Post-processing

This remains as `before.`

1.10 Create flux elements

This remains as `before.`

1.11 Source files for this tutorial

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

demo_drivers/poisson/two_d_poisson_flux_bc_adapt/

- The driver code is:

demo_drivers/poisson/two_d_poisson_flux_bc_adapt/two_d_poisson_flux_bc_adapt.cc

---

1.12 PDF file

A pdf version of this document is available.