# Chapter 1

# Example problem: Adaptive solution of the 2D driven cavity problem

In a previous example we demonstrated the solution of the 2D driven cavity problem using oomph-lib's 2D Taylor-Hood and Crouzeix-Raviart Navier-Stokes elements on a uniform mesh. The computed solution was clearly under-resolved near the corners of the domain where the discontinuity in the velocity boundary conditions creates pressure singularities.

In this example we shall re-solve the driven cavity problem with the refineable versions of oomph-lib's quadrilateral Navier-Stokes elements – the RefineableQTaylorHoodElement<2> and the RefineableQCrouzeixRaviartElement<2>. Enabling spatial adaptivity for this problem involves the same straightforward steps as for a scalar problem:

- The domain must be discretised with a refineable mesh, i.e. a mesh that is derived from the RefineableMesh class.
- An ErrorEstimator object must be specified.

Two additional steps tend to be required during the adaptive solution of Navier-Stokes problems:

- Recall that in Navier-Stokes problems in which the velocity is prescribed along the entire domain boundary, the pressure is only determined up an arbitrary constant, making it necessary to "pin" one pressure value. If the "pinned" pressure degree of freedom is associated with an element that is unrefined during the mesh adaptation, the "pinned " degree of freedom may no longer exist in the adapted problem. To ensure that exactly one pressure degree of freedom is pinned when re-solving the adapted problem, we recommend using the function Problem::actions_after_adapt() to
  1. unpin all pressure values, e.g. using the function
     NavierStokesEquations<DIM>::unpin_all_pressure_dofs(...)
  2. pin a pressure degree of freedom that is known to exist (e.g. the first pressure degree of freedom in the first element of the mesh – whichever element this may be), e.g. using the function
     NavierStokesEquations<DIM>::fix_pressure(...)
- The possible presence of hanging nodes in an adapted mesh requires special treatment for elements (e.g. Taylor-Hood elements) in which the pressure is represented by a low-order interpolation between a subset of the element's nodal values. The required tasks are performed by the function
  NavierStokesEquations<DIM>::pin_redundant_nodal_pressures(...)

  which should be called
  1. before assigning the equation numbers for the first time, and
  2. after every mesh adaptation.

  [If the user "forgets" to call this function, a warning is issued if the library is compiled with the PARANOID flag.]

The driver code discussed below illustrates the use of these functions. The section Comments and Exercises provides a more detailed discussion of the technical details involved in their implementation.

## 1.1 The example problem

We shall illustrate the spatially adaptive solution of the steady 2D Navier-Stokes equations by re-considering the 2D steady driven cavity problem:

**The 2D steady driven cavity problem in a square domain.**

Solve

$$Re\ u_j \frac{\partial u_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \qquad (1)$$

and

$$\frac{\partial u_i}{\partial x_i} = 0,$$

in the square domain $D = \{x_i \in [0,1]; i = 1,2\}$, subject to the Dirichlet boundary conditions

$$\mathbf{u}|_{\partial D} = (0,0), \qquad (2)$$

on right, top and left boundaries and

$$\mathbf{u}|_{\partial D} = (1,0), \qquad (3)$$

on the bottom boundary, $x_2 = 0$ .

### 1.1.1 Solution with Crouzeix-Raviart elements

The figure below shows "carpet plots" of the velocity and pressure fields as well as a contour plot of the pressure distribution with superimposed streamlines for a Reynolds number of $Re = 100$. The velocity vanishes along the entire domain boundary, apart from the bottom boundary $(x_2 = 0)$ where the moving "lid" imposes a unit tangential velocity which drives a large vortex, centred at $(x_1, x_2) \approx (0.62, 0.26)$. The pressure singularities created by the velocity discontinuities at $(x_1, x_2) = (0,0)$ and $(x_1, x_2) = (1,0)$ are now much better resolved.

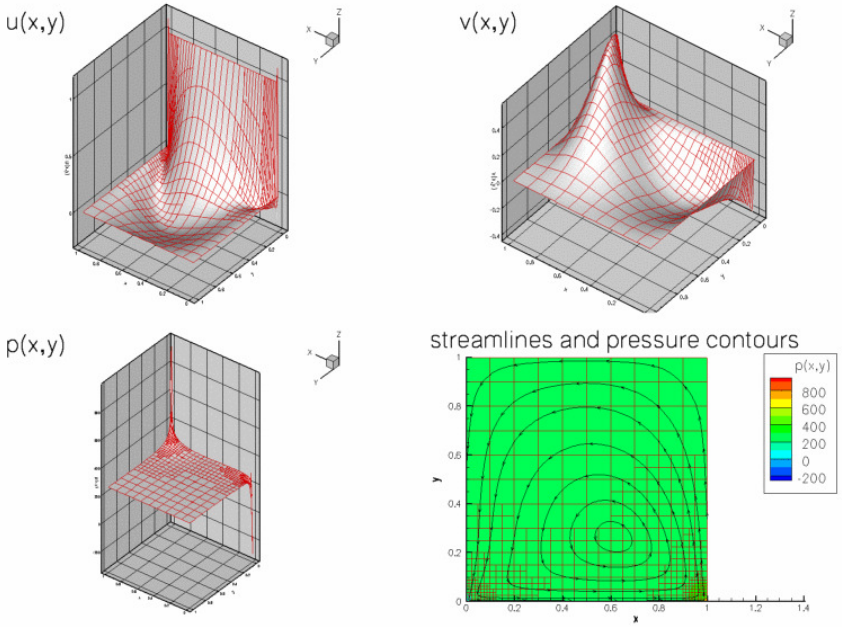


**Figure 1.1 Plot of the velocity and pressure fields computed with adaptive Crouzeix-Raviart elements for Re=100.**

### 1.1.2 Solution with Taylor-Hood elements

The next figure shows the corresponding results obtained from a computation with adaptive Taylor-Hood elements.

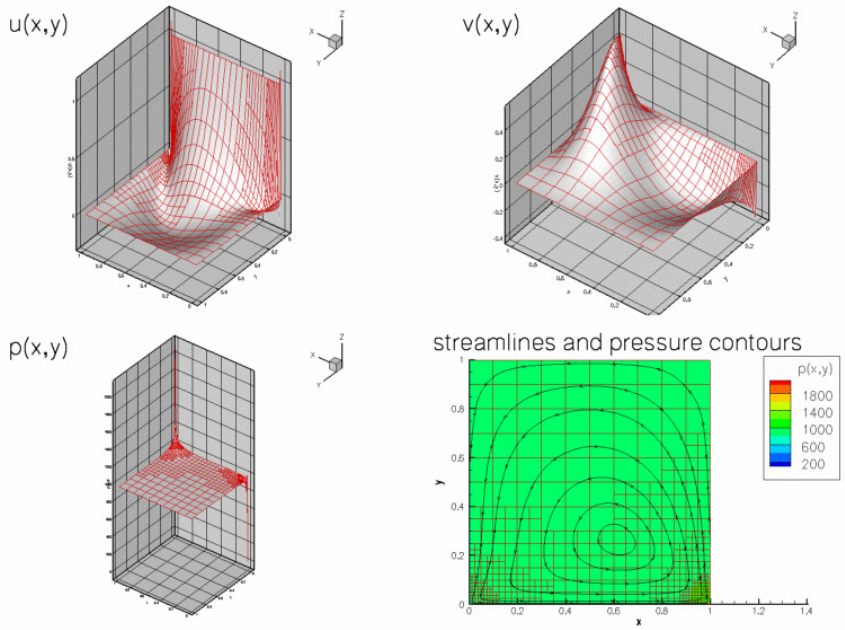


**Figure 1.2 Plot of the velocity and pressure fields computed with adaptive Taylor-Hood elements for Re=100.**

## 1.2 Global parameters and functions

The global namespace used to define the problem parameters is identical to the one in the non-adaptive version.

```
//==start_of_namespace========================================================
/// Namespace for physical parameters
//============================================================================
namespace Global_Physical_Variables
{
 /// Reynolds number
 double Re=100;
} // end_of_namespace
```

## 1.3 The driver code

The main driver code is virtually identical to that in the non-adaptive version. We specify the appropriate refineable element types and use the black-box adaptive Newton solver, allowing for up to three levels of spatial adaptivity.

```
//==start_of_main============================================================
/// Driver for RefineableDrivenCavity test problem
//============================================================================
int main()
{
 // Set output directory
 DocInfo doc_info;
 doc_info.set_directory("RESLT");
 // Set max. number of black-box adaptation
 unsigned max_adapt=3;
 // Solve problem with Taylor Hood elements
 //-------------------------------------
 {
  //Build problem
  RefineableDrivenCavityProblem<RefineableQTaylorHoodElement<2> > problem;

  // Solve the problem with automatic adaptation
  problem.newton_solve(max_adapt);
```

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

  //Output solution
  problem.doc_solution(doc_info);
 } // end of Taylor Hood elements

 // Solve problem with Crouzeix Raviart elements
 //----------------------------------------------
 {
  // Build problem
  RefineableDrivenCavityProblem<RefineableQCrouzeixRaviartElement<2> > problem;

  // Solve the problem with automatic adaptation
  problem.newton_solve(max_adapt);

  // Step number
  doc_info.number()=1;
  //Output solution
  problem.doc_solution(doc_info);
 } // end of Crouzeix Raviart elements
} // end_of_main
```

## 1.4 The problem class

Most of the problem class is identical to that in the non-adaptive version of the code : We provide a constructor and destructor and use the function `Problem::actions_before_newton_solve()` to (re-)assign the boundary conditions.

```
//==start_of_problem_class============================================
/// Driven cavity problem in rectangular domain, templated
/// by element type.
//====================================================================
template<class ELEMENT>
class RefineableDrivenCavityProblem : public Problem
{
public:

 /// Constructor
 RefineableDrivenCavityProblem();

 /// Destructor: Empty
 ~RefineableDrivenCavityProblem() {}

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

 /// Update the problem specs before solve.
 /// (Re-)set velocity boundary conditions just to be on the safe side...
 void actions_before_newton_solve()
  {
  // Setup tangential flow along boundary 0:
  unsigned ibound=0;
  unsigned num_nod= mesh_pt()->nboundary_node(ibound);
  for (unsigned inod=0;inod<num_nod;inod++)
   {
    // Tangential flow
    unsigned i=0;
    mesh_pt()->boundary_node_pt(ibound,inod)->set_value(i,1.0);
    // No penetration
    i=1;
    mesh_pt()->boundary_node_pt(ibound,inod)->set_value(i,0.0);
   }

  // Overwrite with no flow along all other boundaries
  unsigned num_bound = mesh_pt()->nboundary();
  for(unsigned ibound=1;ibound<num_bound;ibound++)
   {
    unsigned num_nod= mesh_pt()->nboundary_node(ibound);
    for (unsigned inod=0;inod<num_nod;inod++)
     {
      for (unsigned i=0;i<2;i++)
       {
        mesh_pt()->boundary_node_pt(ibound,inod)->set_value(i,0.0);
       }
     }
   }
  } // end_of_actions_before_newton_solve
```

As discussed in the introduction, we use the function `Problem::actions_after_adapt()` to ensure that, regardless of the mesh adaptation pattern, exactly one pressure degree of freedom is pinned. We start by unpinning all pressure degrees of freedom:

```
 /// After adaptation: Unpin pressure and pin redudant pressure dofs.
 void actions_after_adapt()
```

```
  {
   // Unpin all pressure dofs
   RefineableNavierStokesEquations<2>::
    unpin_all_pressure_dofs(mesh_pt()->element_pt());
```

[Note that this function (implemented as a static member function of the `NavierStokesEquations<DIM>` class) unpins the pressure degrees of freedom in all elements that are specified by the input argument (a vector of pointers to the these elements). This implementation allows certain elements in a mesh to be excluded from the procedure; this is required in problems where a mesh contains multiple element types. In the present problem, the mesh contains only Navier-Stokes elements, so we pass a vector of pointers to all elements in the mesh (returned by the function `Mesh::element_pt()`] to the function.]

Following the mesh adaptation any redundant nodal pressures must be pinned, so that hanging pressure degrees of freedom are treated correctly. We note that calling this function is essential for Taylor-Hood elements. The function may be executed without any adverse effect for all other Navier-Stokes elements; see Comments and Exercises for more details on the implementation.

```
    // Pin redundant pressure dofs
   RefineableNavierStokesEquations<2>::
    pin_redundant_nodal_pressures(mesh_pt()->element_pt());
```

Finally, we pin a single pressure degree of freedom (the first pressure value in the first element in the mesh) and set its value to zero.

```
   // Now set the first pressure dof in the first element to 0.0
   fix_pressure(0,0,0.0);
  } // end_of_actions_after_adapt
```

The remainder of the problem class remains as `before`.

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


private:

 /// Fix pressure in element e at pressure dof pdof and set to pvalue
 void fix_pressure(const unsigned &e, const unsigned &pdof,
                   const double &pvalue)
  {
   //Cast to proper element and fix pressure
   dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e))->
                          fix_pressure(pdof,pvalue);
  } // end_of_fix_pressure
}; // end_of_problem_class
```

## 1.5 The problem constructor

The constructor remains largely as `before`. We create an adaptive mesh, build and assign an error estimator and pin the redundant nodal pressure degrees of freedom.

```
//==start_of_constructor==============================================
/// Constructor for RefineableDrivenCavity problem
///
//====================================================================
template<class ELEMENT>
RefineableDrivenCavityProblem<ELEMENT>::RefineableDrivenCavityProblem()
{
 // Setup mesh
 // # of elements in x-direction
 unsigned n_x=10;
 // # of elements in y-direction
 unsigned n_y=10;
 // Domain length in x-direction
 double l_x=1.0;
 // Domain length in y-direction
 double l_y=1.0;
 // Build and assign mesh
 Problem::mesh_pt() =
  new RefineableRectangularQuadMesh<ELEMENT>(n_x,n_y,l_x,l_y);
 // Set error estimator
 Z2ErrorEstimator* error_estimator_pt=new Z2ErrorEstimator;
 dynamic_cast<RefineableRectangularQuadMesh<ELEMENT>*>(mesh_pt())->
  spatial_error_estimator_pt()=error_estimator_pt;

 // Set the boundary conditions for this problem: All nodes are
 // free by default -- just pin the ones that have Dirichlet conditions
 // here: All boundaries are Dirichlet boundaries.
 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++)
```

```
    {
     // Loop over values (u and v velocities)
     for (unsigned i=0;i<2;i++)
      {
       mesh_pt()->boundary_node_pt(ibound,inod)->pin(i);
      }
    }
  } // end loop over boundaries
 //Find number of elements in mesh
 unsigned n_element = mesh_pt()->nelement();
 // Loop over the elements to set up element-specific
 // things that cannot be handled by constructor: Pass pointer to Reynolds
 // number
 for(unsigned e=0;e<n_element;e++)
  {
   // Upcast from GeneralisedElement to the present element
   ELEMENT* el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(e));
   //Set the Reynolds number, etc
   el_pt->re_pt() = &Global_Physical_Variables::Re;
  } // end loop over elements

 // Pin redudant pressure dofs
 RefineableNavierStokesEquations<2>::
  pin_redundant_nodal_pressures(mesh_pt()->element_pt());

  // Now set the first pressure dof in the first element to 0.0
 fix_pressure(0,0,0.0);

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

} // end_of_constructor
```

## 1.6 Post-processing

The post-processing function is identical to that in the non-adaptive version of the code.

```
//==start_of_doc_solution=============================================
/// Doc the solution
//====================================================================
template<class ELEMENT>
void RefineableDrivenCavityProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
 ofstream some_file;
 char filename[100];
 // Number of plot points
 unsigned 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_solution
```

## 1.7 Comments and Exercises

### 1.7.1 Hanging nodes in problems with vector-valued unknowns

We discussed in an earlier example for a scalar ( Poisson) problem how oomph-lib's mesh adaptation routines create hanging nodes and how the values that are stored at such nodes are automatically constrained to ensure the inter-element continuity of the solution. The methodology employed for scalar problems is easily generalised to problems with vector-valued unknowns, provided that all unknowns are represented by the same isoparametric interpolation between the elements' nodal values. In such problems, the unknown nodal values at the hanging nodes are constrained to be linear combination of the corresponding values at their "master nodes". The list of "master nodes" and the corresponding "hanging weights" (contained in a HangInfo object) are the same for all unknowns.

To allow the use spatial adaptivity for elements in which different unknowns are represented by different interpolation schemes (e.g. in 2D quadrilateral Taylor-Hood elements where the two velocity components are represented by bi-quadratic interpolation between the values stored at the element's 3x3 nodes, whereas the pressure is represented by bi-linear interpolation between the pressure values stored at the element's 2x2 corner nodes) oomph-lib allows the different nodal values to have their own list of "master nodes" and "hanging weights". This is achieved as follows;

- By default, all nodes are assumed to be non-hanging. This status is indicated by the fact that a Node's

pointer to its `HangInfo` object, accessible via its member function `Node::hanging_pt()`, is `NULL`.

- Mesh adaptation may turn a node into a hanging node. A node's "hanging status" is primarily a geometrical/topological property. If a `Node` is found to be hanging, `oomph-lib's` mesh adaptation procedures create a `HangInfo` object that stores the list of the hanging node's "master nodes" and their respective weights. A pointer to the `HangInfo` object is then passed to the `Node`. The list of master nodes and weights, stored in this `HangInfo` object is then used by the function `Node::position()` to determine the `Node's` (constrained) position.

- Nodes also provide storage for separate pointers to `HangInfo` objects for each of their nodal values. These are accessible via the member function `Node::hanging_pt(i)` which returns the pointer to the `HangInfo` object associated with the node's i-th nodal value. By default, these pointers point to the "geometric" `HangInfo` object, accessible via the argument-free version of this function. This default behaviour is appropriate for isoparametric elements in which all unknowns are represented by interpolation between the elements' nodes, using its geometric shape functions as basis functions.

- For elements that use different interpolation schemes for different nodal values (e.g. in Taylor-Hood elements), the default assignment for the pointers to the `HangInfo` objects may be over-written. This is task is typically performed by re-implementing (and thus over-writing) the empty virtual function
  `RefineableElement::further_setup_hanging_nodes()`

  for such elements. This function is called automatically during at the end of `oomph-lib's` mesh adaptation procedures.

  .

## 1.7.2 Adaptivity for Taylor-Hood elements

### 1.7.2.1 The issues

In non-adaptive 2D [3D] Taylor-Hood elements, every node stores (at least) two [three] nodal values which represent the two [three] velocity components. The four corner [eight vertex] nodes store an additional third [fourth] value which represents the pressure. If the mesh is subjected to non-uniform refinement, some of the mid-side nodes in large elements also act as corner nodes for adjacent smaller elements, as illustrated in the figure below.

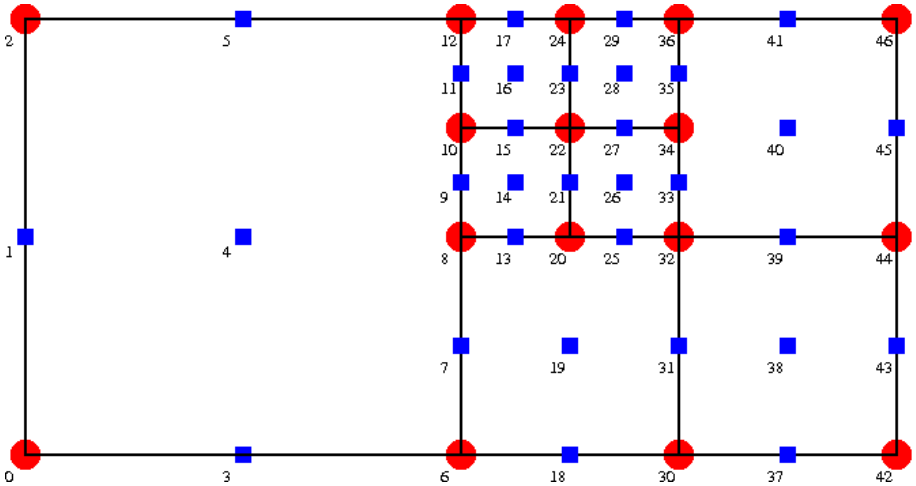


**Figure 1.3 An adapted mesh, with nodes storing a pressure value denoted by circles, and velocity nodes by squares.**

The figure illustrates that the "hanging status" of the various degrees of freedom can be become fairly involved. For instance

- Node 7 is geometrically hanging, with master nodes 6, 8 and 12. It is not a pressure node.

- Node 8 is geometrically non-hanging, but it is a hanging node for the pressure interpolation. Its pressure master nodes are 6 and 12.

- Node 10 is geometrically hanging, and its geometric master nodes are 6, 8 and 12, while its pressure master nodes are 6 and 12.

To illustrate that `oomph-lib's` automatic mesh adaptation procedures are able to deal with these cases, the figure below shows a "carpet plot" of the pressure distribution, $p(x_1, x_2)$, obtained from a (strongly under-resolved) driven-cavity computation on the mesh shown above. The figure illustrates that the hanging node constraints ensure the inter-element continuity of the pressure throughout the domain, and, in particular, along the "right" boundary of the largest element.

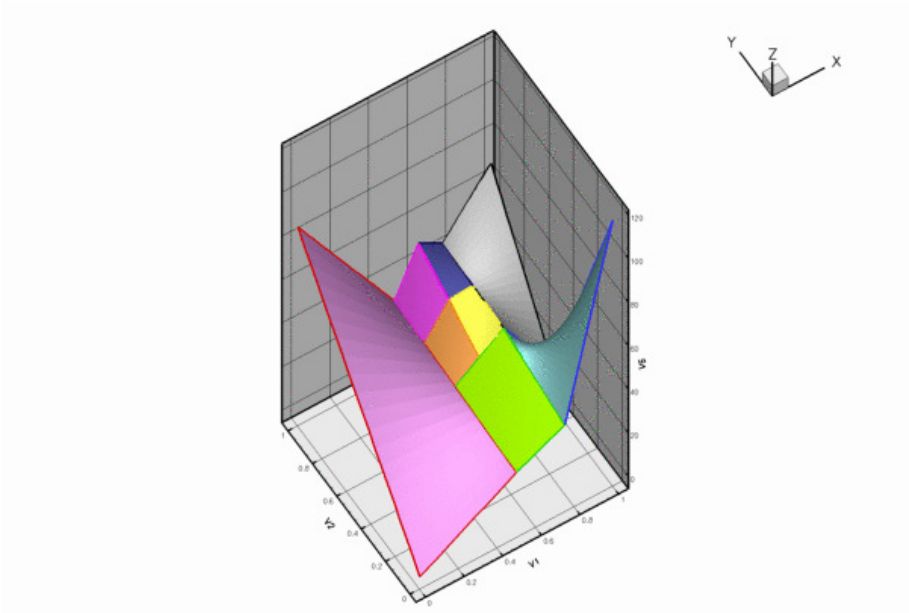


**Figure 1.4 The under-resolved pressure distribution for the driven cavity problem, using the above mesh and computed with Taylor-Hood elements.**

#### 1.7.2.2 Details of the implementation

To facilitate the book-keeping for such problems, *all* nodes in the refineable Taylor-Hood elements store three [four] nodal values, even though, depending on the mesh's refinement pattern some of the pressure values will not be used. To eliminate the "redundant" pressure degrees of freedom from the problem, we provide the function
`NavierStokesEquations<DIM>::pin_redundant_nodal_pressures(...)`
which pins the "redundant" pressure degrees of freedom in all elements specified by the input argument (a vector of pointers to the Navier-Stokes elements). This function must be called after the initial mesh has been created, and after each mesh adaptation. The function first pins *all* nodal pressure values, using the function
`NavierStokesEquations<DIM>::pin_all_nodal_pressure_dofs()`
and then unpins the nodal pressure values at the elements' corner [vertex] nodes, using the function
`NavierStokesEquations<DIM>::unpin_proper_nodal_pressure_dofs()`
These functions are implemented as empty virtual functions in the `NavierStokesEquations<DIM>` class which provides a base class for all Navier-Stokes elements. The empty functions are overwritten for `QTaylor↩ HoodElement<DIM>` and remain empty for all other Navier-Stokes elements, therefore they can be called for any element type.

### 1.7.3 Adaptivity for Crouzeix-Raviart elements

As discussed in the `previous example`, `oomph-lib's` isoparametric 2D [3D] Crouzeix-Raviart elements employ a piecewise bi- [tri-]linear, globally discontinuous pressure representation. In each element, the pressure is represented by bi-[tri-]linear basis functions, multiplied by 3 [4] pressure values which are stored in the element's internal `Data`. Since the pressure representation is discontinuous, the pressure values do not have to be subjected to any constraints to ensure inter-element continuity. Each `Node` stores two [three] velocity degrees of freedom. Since the velocity representation is isoparametric, the default assignment for the nodal values' `HangInfo` pointer is appropriate and no further action is required.

### 1.7.4 Exercises

1. Confirm that a warning message is issued if the function `NavierStokesEquations<DIM>::pin_all_nodal_pressure_dofs()` is not called following the mesh adaptation.
2. Investigate how the pressure distribution changes with each adaptation. [Hint: You can call `doc_solution(...)` from `actions_after_newton_solve()` to document the progress of the mesh adaptation.]

---

## 1.8 Source files for this tutorial

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

  `demo_drivers/navier_stokes/adaptive_driven_cavity/`

- The driver code is:

  `demo_drivers/navier_stokes/adaptive_driven_cavity/adaptive_driven_cavity.cc`

---

## 1.9 PDF file

A pdf version of this document is available.