Chapter 1

Demo problem: Fluid Mechanics on unstructured 3D meshes

This tutorial provides another demonstration of how to use 3D unstructured meshes for the solution of fluid flow problems. (The main `tetgen tutorial` already contains a 3D unstructured fluid example.)

The specific problem considered here also serves as a "warm-up problem" for the `corresponding fluid-structure interaction problem` in which the domain boundary is replaced by an elastic vessel.

1.1 The problem

Here is a sketch of the problem: Flow is driven through a 3D rigid vessel made of three rectangular tubes that meet at a common junction. The flow is driven by a prescribed pressure drop between the upstream and the two downstream ends, $\Delta P^* = P^*_{in} - P^*_{out}$, and we assume/impose parallel in- and outflow in the inlet and outlet cross-sections, all of which are parallel to $x - y$ coordinate plane.

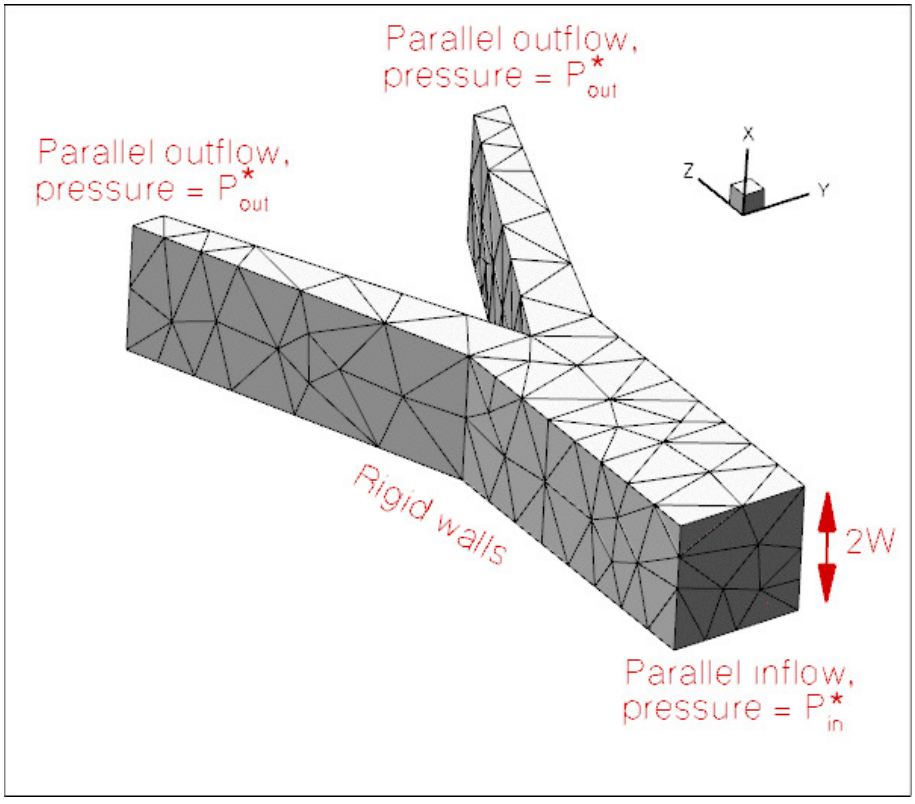

Figure 1.1 Sketch of the domain with boundary conditions.

We non-dimensionalise all lengths on the half-width, W, of the square main vessel and use the overall pressure drop, ΔP^* to define the (viscous) velocity scale

$$\mathcal{U} = \frac{\Delta P^* \, W}{\mu}.$$

With this choice the Reynolds number becomes

$$Re = \frac{\rho \mathcal{U} W}{\mu} = \frac{\Delta P^* \rho W^2}{\mu^2},$$

and we choose to drive the flow with a dimensionless pressure drop of $\Delta P = 1$. An increase in Reynolds number may therefore be interpreted as in increase in the applied (dimensional) pressure drop along the vessel.

1.2 3D unstructured mesh generation

We use `Hang Si's` open-source mesh generator `tetgen` to generate the unstructured tetrahedral mesh "offline". We then process the output files produced by `tetgen` to generate an unstructured `oomph-lib` mesh.

`Tetgen` requires the specification of the domain boundaries via so-called facets – planar surface patches that are bounded by closed polygonal line segments. In our simple geometry each of the three tube segments has four external faces. Together with the three in- and outflow sections this results in a total of 15 facets.

The 15 facets are defined in a `*.poly` file that specifies the position of the vertices, and identifies the facets via a "face list" that establishes their bounding vertices. The well-annotated `*.poly` file for this problem is located at:

`demo_drivers/navier_stokes/unstructured_three_d_fluid/fsi_bifurcation_fluid.poly`

We refer to the `tetgen webpages` and `oomph-lib's` own `tetgen tutorial` for further details on how to create `*.poly` files.

Here is a plot of the domain specified by `fsi_bifurcation_fluid.poly`. The plot was created using `tetview` which is distributed with `tetgen` .

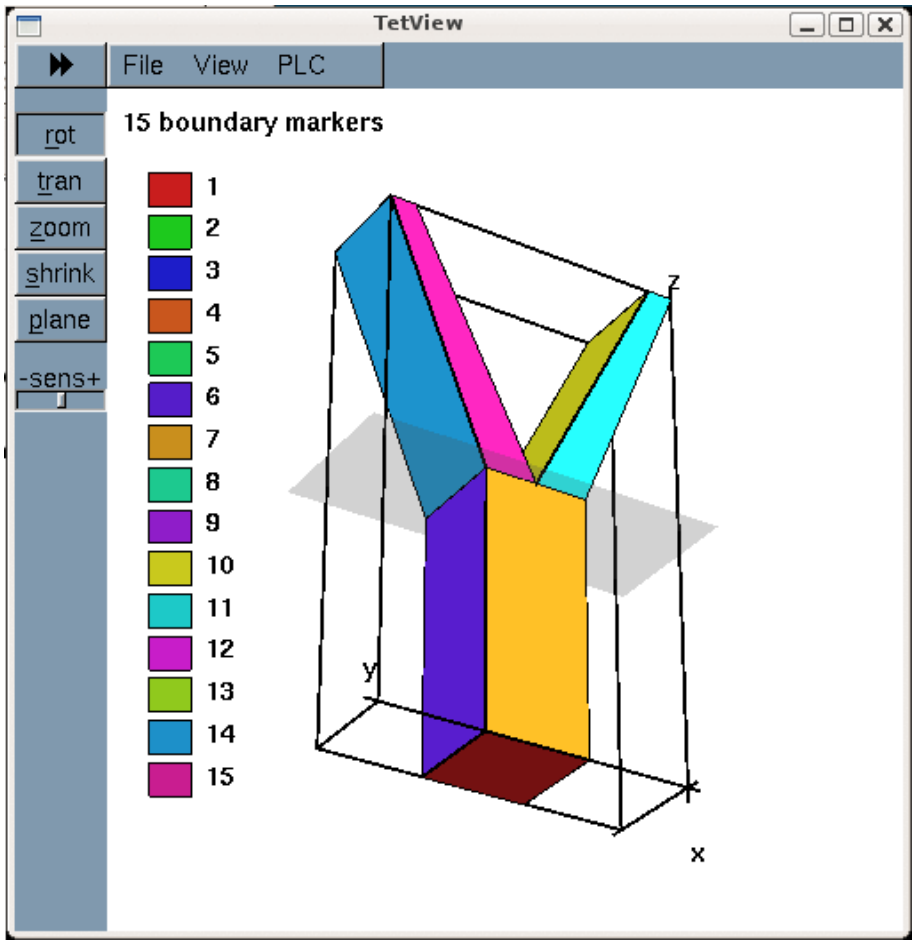

Figure 1.2 The domain and its bounding facets.

Note that we have deliberately assigned a different boundary ID to each facet. This will make the assignment of the boundary condition somewhat tedious as the domain boundaries of interest tend to be represented by multiple, separate mesh boundaries. However, the assignment of distinct boundary IDs for the different facets is essential for the automatic generation of boundary coordinates in the corresponding fluid-structure interaction problem and is therefore **strongly recommended**.

Tetgen generates an unstructured volumetric mesh from the information contained in the `*.poly` file and outputs the mesh's nodes, elements and faces in the files

- demo_drivers/navier_stokes/unstructured_three_d_fluid/fsi_bifurcation_fluid.1.node
- demo_drivers/navier_stokes/unstructured_three_d_fluid/fsi_bifurcation_fluid.1.ele
- demo_drivers/navier_stokes/unstructured_three_d_fluid/fsi_bifurcation_fluid.1.face

These files can be used as input to `oomph-lib's TetgenMesh` class, using the procedure discussed in another tutorial.

The figure below shows a `tetview` plot of the mesh, created with a volume constraint of 0.2 (i.e. the maximum volume of each tetrahedron is guaranteed to be less than 0.2 units), using the command

```
tetgen -a0.2 fsi_bifurcation_fluid.poly
```

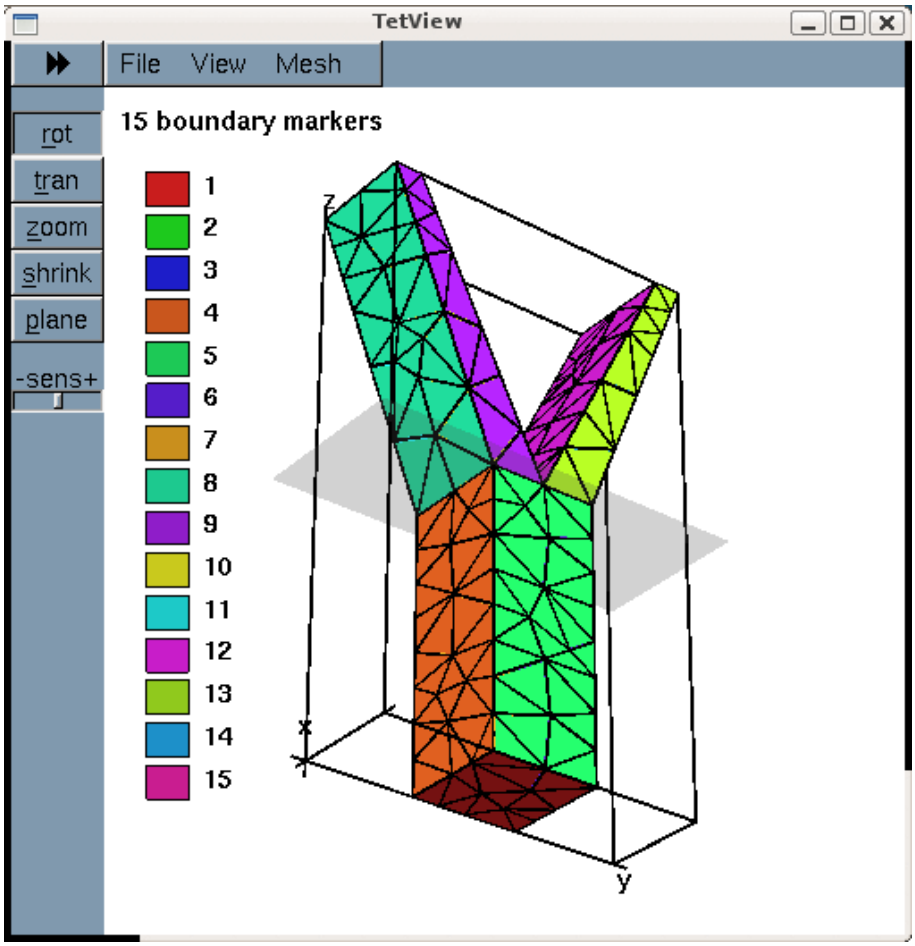

Figure 1.3 Plot of the mesh, generated by tetgen.

Note how `tetgen` has subdivided each of the 15 original facets specified in the `*.poly` file into a surface triangulation. The nodes and tetrahedral elements that are located on (or adjacent to) the 15 original facets inherit their boundary IDs. This is important when we assign the boundary conditions for the actual computation.

1.3 Results

The plot shown below illustrates the flow field (streamribbons coloured by pressure contours) for a Reynolds number of $Re = 100$. The transparent faces show the boundaries of the fluid elements and illustrate that the mesh is very coarse. As a result, the flow is clearly under-resolved, particularly near the two outflow cross-sections where the imposition of parallel outflow forces the fluid velocity to re-adjust rapidly as it approaches the outlet. (See Creating a finer mesh and applying more appropriate outflow boundary conditions in Comments and Exercises for a more detailed discussion of this aspect.)

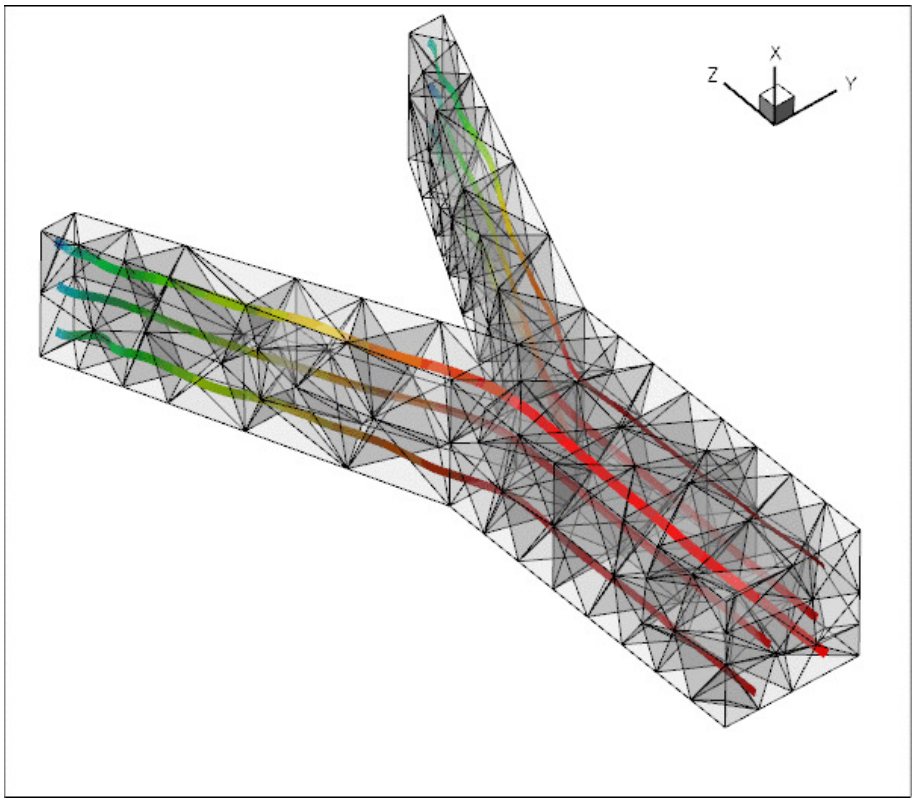

Figure 1.4 Flowfield (streamribbons, coloured by the pressure contours) and element boundaries.

1.4 Problem parameters

As usual we define the various problem parameters in a global namespace. We define the Reynolds number and specify the tractions to be applied at the in- and outflow cross-sections:

```
//=======start_namespace==========================================
/// Global variables
//================================================================
namespace Global_Parameters
{

 /// Default Reynolds number
 double Re=100.0;

 /// Fluid pressure on inflow boundary
 double P_in=0.5;

 /// Applied traction on fluid at the inflow boundary
 void prescribed_inflow_traction(const double& t,
                                 const Vector<double>& x,
                                 const Vector<double>& n,
                                 Vector<double>& traction)
 {
  traction[0]=0.0;
  traction[1]=0.0;
  traction[2]=P_in;
 }

 /// Fluid pressure on outflow boundary
 double P_out=-0.5;

 /// Applied traction on fluid at the inflow boundary
 void prescribed_outflow_traction(const double& t,
                                  const Vector<double>& x,
                                  const Vector<double>& n,
                                  Vector<double>& traction)
 {
  traction[0]=0.0;
  traction[1]=0.0;
  traction[2]=-P_out;
 }

} //end namespace
```

1.5 The driver code

We specify an output directory, create the `Problem` object using ten-node tetrahedral Taylor-Hood elements, and output the initial guess for the flow field:

```
//=============start_main=====================================================
/// Demonstrate how to solve an unstructured 3D fluids problem
//============================================================================
int main(int argc, char **argv)
{
 // Store command line arguments
 CommandLineArgs::setup(argc,argv);

 // Label for output
 DocInfo doc_info;

 // Parameter study
 double Re_increment=100.0;
 unsigned nstep=4;
 if (CommandLineArgs::Argc==2)
  {
   std::cout << "Validation -- only doing two steps" << std::endl;
   nstep=2;
  }


 //Taylor--Hood
 {
  // Output directory
  doc_info.set_directory("RESLT_TH");

  //Set up the problem
  UnstructuredFluidProblem<TTaylorHoodElement<3> > problem;

  //Output initial guess
  problem.doc_solution(doc_info);
  doc_info.number()++;
```

Next we perform a parameter study in which we increase the Reynolds number of the flow – corresponding to an increase in the applied pressure drop. (As usual we perform a smaller number of steps in a validation run – performed when the code is run with nonzero number of command-line arguments.)

```
  // Parameter study: Crank up the pressure drop along the vessel
  for (unsigned istep=0;istep<nstep;istep++)
   {
    // Solve the problem
    problem.newton_solve();

    //Output solution
    problem.doc_solution(doc_info);
    doc_info.number()++;

    // Bump up Reynolds number (equivalent to increasing the imposed pressure
    // drop)
    Global_Parameters::Re+=Re_increment;
   }
 }
 //Crouzeix Raviart
 {
  //Reset to default Reynolds number
  Global_Parameters::Re = 100.0;
  //Reset doc info number
  doc_info.number()=0;
  // Output directory
  doc_info.set_directory("RESLT_CR");

  //Set up the problem
  UnstructuredFluidProblem<TCrouzeixRaviartElement<3> > problem;
  //Output initial guess
  problem.doc_solution(doc_info);
  doc_info.number()++;

  // Parameter study: Crank up the pressure drop along the vessel
  for (unsigned istep=0;istep<nstep;istep++)
   {
    // Solve the problem
    problem.newton_solve();

    //Output solution
    problem.doc_solution(doc_info);
    doc_info.number()++;

    // Bump up Reynolds number (equivalent to increasing the imposed pressure
    // drop)
    Global_Parameters::Re+=Re_increment;
   }
```

```
 }

} // end_of_main
```

1.6 The Problem class

The `Problem` class has the usual member functions and provides explicit storage for the fluid mesh and the meshes containing the `FaceElements` that apply the traction conditions at the in- and outflow boundaries. We also provide storage for the IDs of the mesh boundaries that constitute the in- and outflow boundaries to facilitate the application of the boundary conditions.

```
//======start_problem_class============================================
/// Unstructured fluid problem
//====================================================================
template<class ELEMENT>
class UnstructuredFluidProblem : public Problem
{
public:

 /// Constructor:
 UnstructuredFluidProblem();

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

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

 /// Return total number of fluid inflow traction boundaries
 unsigned nfluid_inflow_traction_boundary()
  {
   return Inflow_boundary_id.size();
  }

 /// Return total number of fluid outflow traction boundaries
 unsigned nfluid_outflow_traction_boundary()
  {
   return Outflow_boundary_id.size();
  }

 /// Return total number of fluid outflow traction boundaries
 unsigned nfluid_traction_boundary()
  {
   return Inflow_boundary_id.size()+Outflow_boundary_id.size();
  }
 //private:

 /// Create fluid traction elements at inflow
 void create_fluid_traction_elements();

 /// Bulk fluid mesh
 TetgenMesh<ELEMENT>* Fluid_mesh_pt;

 /// Meshes of fluid traction elements that apply pressure at in/outflow
 Vector<Mesh*> Fluid_traction_mesh_pt;

 /// IDs of fluid mesh boundaries along which inflow boundary conditions
 /// are applied
 Vector<unsigned> Inflow_boundary_id;

 /// IDs of fluid mesh boundaries along which inflow boundary conditions
 /// are applied
 Vector<unsigned> Outflow_boundary_id;
};
```

1.7 The Problem constructor

We start by building the fluid mesh, using the files created by `tetgen` :

```
//==========start_constructor==========================================
/// Constructor for unstructured 3D fluid problem
//====================================================================
template<class ELEMENT>
UnstructuredFluidProblem<ELEMENT>::UnstructuredFluidProblem()
{

 //Create fluid bulk mesh, sub-dividing "corner" elements
 string node_file_name="fsi_bifurcation_fluid.1.node";
 string element_file_name="fsi_bifurcation_fluid.1.ele";
 string face_file_name="fsi_bifurcation_fluid.1.face";
 bool split_corner_elements=true;
 Fluid_mesh_pt =  new TetgenMesh<ELEMENT>(node_file_name,
                                          element_file_name,
```

```
                                        face_file_name,
                                        split_corner_elements);
```

(We refer to the subsection Splitting corner elements in unstructured meshes to avoid locking in the section Comments and Exercises for a discussion of the `split_corner_elements` flag).

Next, we set up the boundary lookup schemes that determine which elements are located next to which domain boundaries, and specify the IDs of the mesh boundaries that coincide with the in- and outflow cross-sections. Note that this information reflects the specification of the boundary IDs in the `tetgen *.poly` file.

```
// Find elements next to boundaries
//Fluid_mesh_pt->setup_boundary_element_info();
// The following corresponds to the boundaries as specified by
// facets in the tetgen input:
// Fluid mesh has one inflow boundary: Boundary 0
Inflow_boundary_id.resize(1);
Inflow_boundary_id[0]=0;

// Fluid mesh has two outflow boundaries: Boundaries 1 and 2
Outflow_boundary_id.resize(2);
Outflow_boundary_id[0]=1;
Outflow_boundary_id[1]=2;
```

Next we apply the boundary conditions. We impose parallel in- and outflow by pinning the transverse velocities at all nodes that are located on the in- and outflow boundaries, identifying the nodes via the boundary IDs just set up. We use the boolean map `done` to indicate which boundaries we have visited already.

```
// Apply BCs
//----------

// Map to indicate which boundary has been done
std::map<unsigned,bool> done;

// Loop over inflow/outflow boundaries to impose parallel flow
for (unsigned in_out=0;in_out<2;in_out++)
 {
  // Loop over in/outflow boundaries
  unsigned n=nfluid_inflow_traction_boundary();
  if (in_out==1) n=nfluid_outflow_traction_boundary();
  for (unsigned i=0;i<n;i++)
   {
    // Get boundary ID
    unsigned b=0;
    if (in_out==0)
     {
      b=Inflow_boundary_id[i];
     }
    else
     {
      b=Outflow_boundary_id[i];
     }
    // Number of nodes on that boundary
    unsigned num_nod=Fluid_mesh_pt->nboundary_node(b);
    for (unsigned inod=0;inod<num_nod;inod++)
     {
      // Get the node
      Node* nod_pt=Fluid_mesh_pt->boundary_node_pt(b,inod);

      // Pin transverse velocities
      nod_pt->pin(0);
      nod_pt->pin(1);
     }

    // Done!
    done[b]=true;
   }
 } // done in and outflow
```

The nodes on all other boundaries (i.e. the ones for which `done`[b] is still `false`) are subjected to no-slip conditions by pinning all three velocity components. (This approach facilitates the "extension" of the mesh discussed in section Creating a finer mesh and applying more appropriate outflow boundary conditions, but we note that, in general, keeping track of the boundary IDs associated with each physical boundary must be done "by hand".)

```
// Loop over all fluid mesh boundaries and pin velocities
// of nodes that haven't been dealt with yet
unsigned nbound=Fluid_mesh_pt->nboundary();
for(unsigned b=0;b<nbound;b++)
 {
  // Has the boundary been done yet?
  if (!done[b])
   {
    unsigned num_nod=Fluid_mesh_pt->nboundary_node(b);
    for (unsigned inod=0;inod<num_nod;inod++)
```

```
      {
       // Get node
       Node* nod_pt= Fluid_mesh_pt->boundary_node_pt(b,inod);

       // Pin all velocities
       nod_pt->pin(0);
       nod_pt->pin(1);
       nod_pt->pin(2);
      }
    }
  } // done no slip elsewhere
```

We complete the build of the Navier-Stokes elements by specifying the pointer to the Reynolds number,

```
// Complete the build of the fluid elements so they are fully functional
//----------------------------------------------------------------------
unsigned n_element = Fluid_mesh_pt->nelement();
for(unsigned e=0;e<n_element;e++)
 {
  // Upcast from GeneralisedElement to the present element
  ELEMENT* el_pt = dynamic_cast<ELEMENT*>(Fluid_mesh_pt->element_pt(e));

  //Set the Reynolds number
  el_pt->re_pt() = &Global_Parameters::Re;
 }
```

and attach the `FaceElements` that apply the imposed in- and outflow tractions to the appropriate faces of the elements on the in- and outflow boundaries:

```
// Create meshes of fluid traction elements at inflow/outflow
//-----------------------------------------------------------

// Create the meshes
unsigned n=nfluid_traction_boundary();
Fluid_traction_mesh_pt.resize(n);
for (unsigned i=0;i<n;i++)
 {
  Fluid_traction_mesh_pt[i]=new Mesh;
 }

// Populate them with elements
create_fluid_traction_elements();
```

Finally, we combine the various sub-meshes to a combined global mesh and assign the equation numbers.

```
// Combine the lot
//----------------

// Add sub meshes:
// Fluid bulk mesh
add_sub_mesh(Fluid_mesh_pt);

// The fluid traction meshes
n=nfluid_traction_boundary();
for (unsigned i=0;i<n;i++)
 {
  add_sub_mesh(Fluid_traction_mesh_pt[i]);
 }

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

} // end constructor
```

1.8 Creating the fluid traction elements

The helper function `create_fluid_traction_elements()` loops over the bulk elements that are adjacent to the in- and outflow cross-sections and attaches `NavierStokesTractionElements` to the relevant faces. We store pointers to the newly-created elements in the appropriate meshes, and pass pointers to the functions that specify the imposed traction to the elements.

```
//============start_of_fluid_traction_elements==============================
/// Create fluid traction elements
//==========================================================================
template<class ELEMENT>
void UnstructuredFluidProblem<ELEMENT>::create_fluid_traction_elements()
{
 // Counter for number of fluid traction meshes
 unsigned count=0;
 // Loop over inflow/outflow boundaries
```

```
 for (unsigned in_out=0;in_out<2;in_out++)
  {
   // Loop over boundaries with fluid traction elements
   unsigned n=nfluid_inflow_traction_boundary();
   if (in_out==1) n=nfluid_outflow_traction_boundary();
   for (unsigned i=0;i<n;i++)
    {
     // Get boundary ID
     unsigned b=0;
     if (in_out==0)
      {
       b=Inflow_boundary_id[i];
      }
     else
      {
       b=Outflow_boundary_id[i];
      }

     // How many bulk elements are adjacent to boundary b?
     unsigned n_element = Fluid_mesh_pt->nboundary_element(b);

     // Loop over the bulk elements adjacent to boundary b
     for(unsigned e=0;e<n_element;e++)
      {
       // Get pointer to the bulk element that is adjacent to boundary b
       ELEMENT* bulk_elem_pt = dynamic_cast<ELEMENT*>(
        Fluid_mesh_pt->boundary_element_pt(b,e));

       //What is the index of the face of the element e along boundary b
       int face_index = Fluid_mesh_pt->face_index_at_boundary(b,e);

       // Create new element
       NavierStokesTractionElement<ELEMENT>* el_pt=
        new NavierStokesTractionElement<ELEMENT>(bulk_elem_pt,
                                                 face_index);

       // Add it to the mesh
       Fluid_traction_mesh_pt[count]->add_element_pt(el_pt);

       // Set the pointer to the prescribed traction function
       if (in_out==0)
        {
         el_pt->traction_fct_pt() =
          &Global_Parameters::prescribed_inflow_traction;
        }
       else
        {
         el_pt->traction_fct_pt() =
          &Global_Parameters::prescribed_outflow_traction;
        }
      }
     // Bump up counter
     count++;
    }
  }

 } // end of create_traction_elements
```

1.9 Post-processing

The post-processing routine outputs the flow field.

```
//========================================================================
/// Doc the solution
//========================================================================
template<class ELEMENT>
void UnstructuredFluidProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
 ofstream some_file;
 char filename[100];
 // Number of plot points
 unsigned npts;
 npts=5;


 // Output fluid solution
 sprintf(filename,"%s/fluid_soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 Fluid_mesh_pt->output(some_file,npts);
 some_file.close();

}
```

1.10 Comments and Exercises

1.10.1 Splitting corner elements in unstructured meshes to avoid locking

Meshes generated by `tetgen` tend to be of very high quality and can usually be used without further modification. However, in Navier-Stokes computations (or other problems in which the elements have to satisfy an LBB-type stability constraint) problems can arise if too many of an element's nodes are constrained by boundary conditions, causing the discretisation to "lock". This tends to happen when three of the element's four faces are located on domain boundaries, as in the case of the top-left and bottom-left elements in the inflow cross-section shown below.

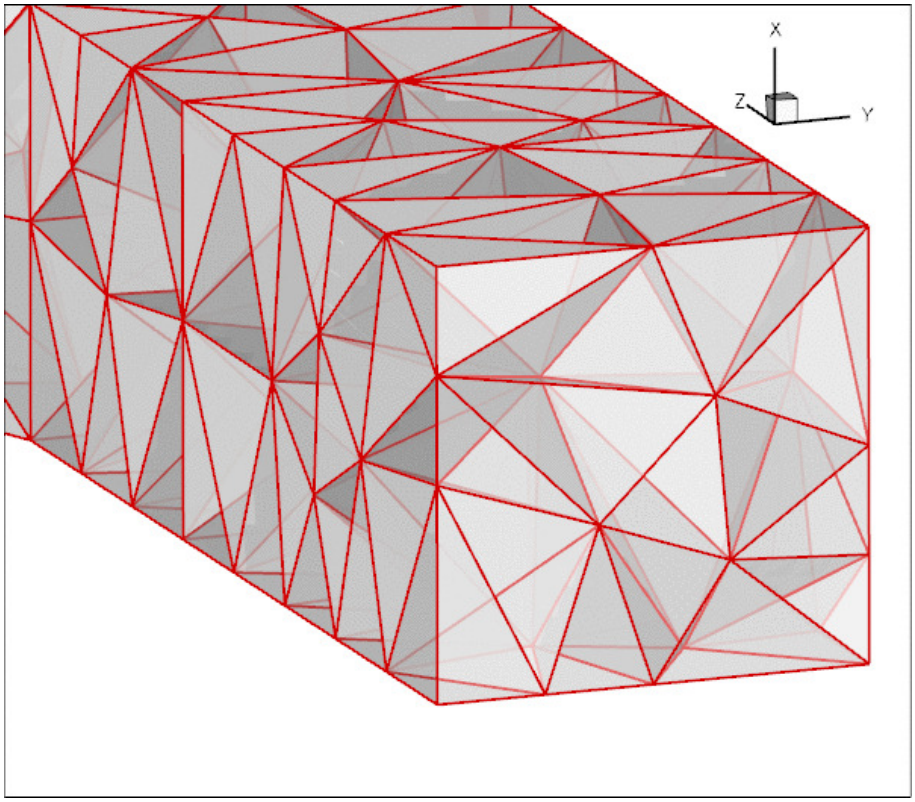

Figure 1.5 Plot of the mesh generated by tetgen.

Ten-noded tetrahedral elements have nodes at their vertices and on their edges only. Therefore all the nodes in these particular two elements are located on domain boundaries. Furthermore, only one of these (the node halfway along the edge that traverses the inflow cross-section) is unconstrained; the imposition of parallel flow at the inflow face only constrains the node's transverse velocities. Thus, both elements have a single velocity degree of freedom (the axial velocity at the node on the inflow face) but they retain their four pressure degrees of freedom (the pressures at their vertices). This does not *necessarily* over-constrain the problem (and in the present problem it does not) – "locking", due to the presence of too many pressure degrees of freedom, arises at the level of the global, fully-assembled problem, not on an element-by-element basis. However, the presence of such elements makes the occurrence of locking more likely and we have occasionally come across examples in which locking does occur. The remedy would be to pin exactly the required number of superfluous pressure degrees of freedom (no more and no fewer!) to "unlock" the problem. However, the diagnosis of the problem is as difficult as its practical resolution: How do we determine in advance if a problem will lock and, if so, which pressures should be pinned, etc.?
To avoid the problem altogether, `oomph-lib's TetgenMesh` constructor provides an optional boolean flag, `split_corner_elements`. If this flag is set to `true`, the mesh constructor identifies all elements in which at least three faces are located on mesh boundaries. These elements are then split into four smaller ones, using a localised refinement as shown in this plot:

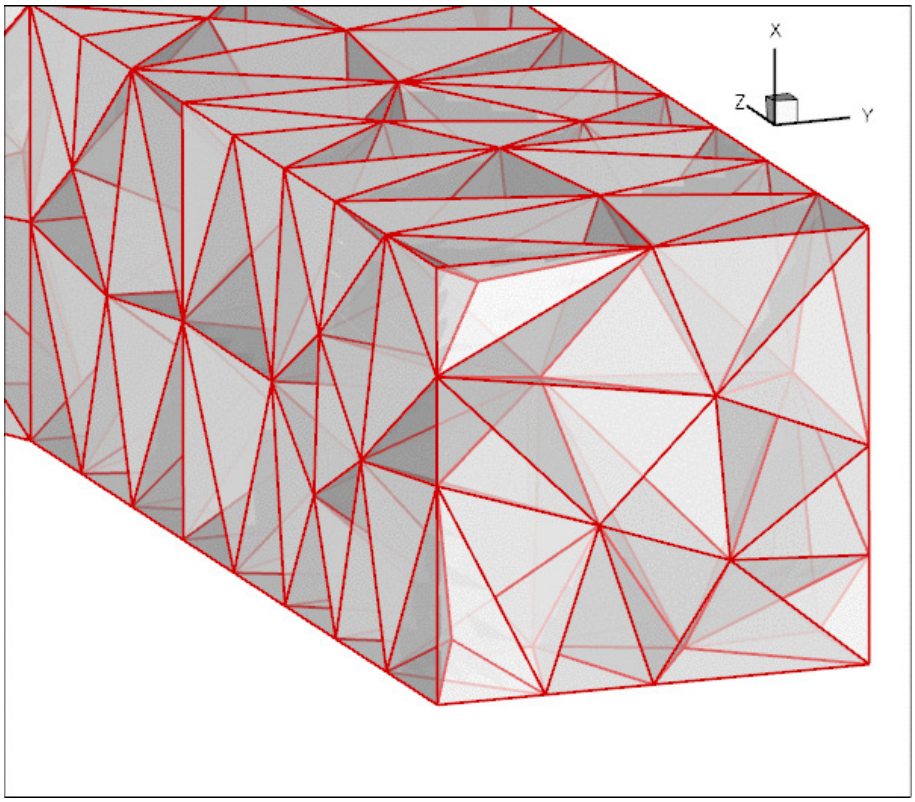

Figure 1.6 Plot of the mesh after splitting corner elements.

Note that the localised refinement leads to a small deterioration in the element quality but this is usually acceptable. If you are concerned about this aspect, allow the splitting of corner elements only if locking actually occurs (see below), or try to generate a different `tetgen` mesh that does not suffer from this problem, e.g. by imposing a different volume constraint.

How to spot "locking":

- If a solution can be computed (using a direct solver) the occurrence of "locking" is easy to spot: Typically the pressure in over-constrained elements becomes extremely large while the rest of the flow field looks fairly normal. Iterative solvers may fail to converge. If you suspect that locking may have occurred, try re-computing the solution on a mesh with the `split_corner_elements` flag set to `true`.

1.10.2 Creating a finer mesh and applying more appropriate outflow boundary conditions

The flow field shown in the Results section is clearly under-resolved and a much finer mesh would have to be used to fully resolve all flow features. The problem is particularly bad because we have imposed parallel outflow (i.e. flow in the z-direction) at the ends of tubes whose axes are not aligned with the z-axis. This creates thin outflow boundary layers within which the flow is forced to change direction as it exits the tubes. These observations motivate the following exercises that allow you to explore unstructured mesh generation.

- **Exercise 1: Refining the mesh**

  Create a finer mesh by specifying a smaller volume constraint for tetgen. For instance, using

  ```
  tetgen -a0.02 fsi_bifurcation_fluid.poly
  ```

  will generate a much finer mesh, containing 3345 tets. Note that the driver code can remain completely unchanged.

- **Exercise 2: Add straight outflow vessels**

  The imposition of parallel flow at the outlet boundaries would be less problematic if at least the final part of the two outflow tubes was aligned with the z-axis. Modify the file `fsi_bifurcation_fluid.poly` so that two straight vessel segments, parallel to the z-axis, are added to the mesh. The modification to the `*.poly` file should be straightforward. You have to add eight additional vertices and faces, and re-define the two outflow faces.

  Here is a sample plot of a modified mesh in which two additional straight segments of different lengths have been attached to the downstream tubes. Note that the enumeration of the in- and outflow boundaries was retained, allowing the mesh to be used with the same driver code.

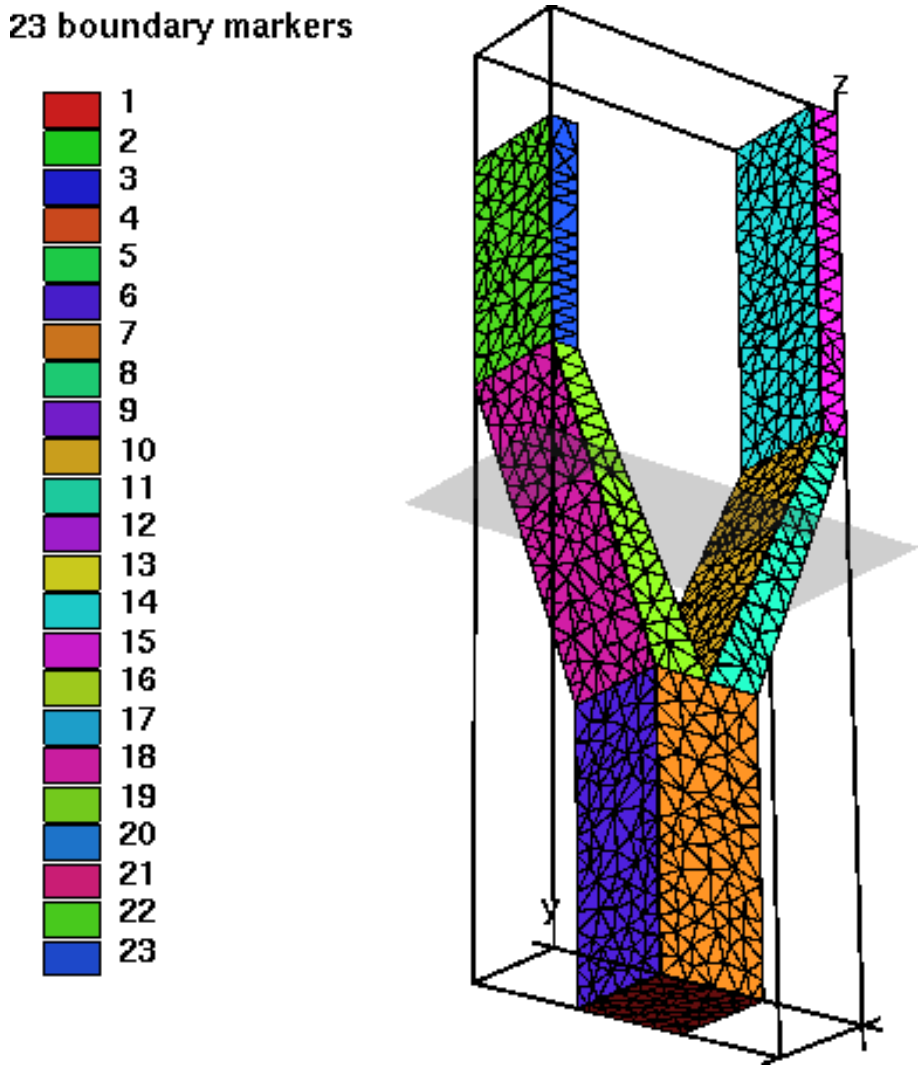

Figure 1.7 Plot of the mesh, with two straight segments added to the outflow branches.

The modified mesh was created with the file `fsi_bifurcation_fluid_with_extended_tubes.poly` which you may wish to consult. However, we do encourage you to do this exercise by yourself first, in order to familiarise yourself with `tetgen` – you will notice that `tetgen` is not very forgiving (or verbose) when it encounters errors in the `*.poly` file.

- **Exercise 3: Use Lagrange multipliers to allow parallel outflow in cross-sections that are not aligned**

with the coordinate axes.

The main (only?) reason why we tend to impose parallel in- or outflow in cross-sections that are aligned with the Cartesian coordinate planes is, of course, that such boundary conditions are easiest to apply in a discretisation that is based on the Cartesian form of the Navier-Stokes equations. What if we wished to impose parallel outflow from cross-sections that are not aligned with such planes? The proper way to do this is to use Lagrange multipliers to enforce the two constraints

$$\mathbf{u} \cdot \mathbf{t}_\alpha = 0 \quad \text{for } \alpha = 1, 2,$$

where $\mathbf{t}_\alpha$ (for $\alpha = 1, 2$) are the two tangent vectors spanning the in- or outflow cross-sections. The imposition of these constraints requires two Lagrange multiplier fields, defined in the in-/outflow cross-sections. Physically, these Lagrange multipliers act as tangential tractions that enforce the parallel in- or outflow. The Lagrange multipliers introduce additional degrees of freedom into the problem and an implementation can be based on an extension of the existing `NavierStokesTractionElements`, using a technique similar to that used to enforce prescribed boundary displacements in solid mechanics problems, discussed in another tutorial. Note that, compared to the previous two exercises, this one is not entirely trivial. An implementation is provided in `ImposeParallelOutflowElement`, but, as always, it is more instructive to try to write it yourself!

1.11 Source files for this tutorial

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

  demo_drivers/navier_stokes/unstructured_three_d_fluid/

- The driver code is:

  demo_drivers/navier_stokes/unstructured_three_d_fluid/unstructured_three_d_fluid.cc

1.12 PDF file

A pdf version of this document is available.