# Chapter 1

# Demo problem: Steady finite-Reynolds-number flow through an iliac bifurcation

The purpose of this tutorial is to demonstrate the simulation of cardiovascular fluid mechanics problems with `oomph-lib`. We showed in `another tutorial` how to use `VMTK` together with `oomph-lib's` conversion code

`demo_drivers/meshing/mesh_from_vmtk/create_fluid_and_solid_surface_mesh_from_fluid_xda_mesh.cc`

to generate unstructured fluid and solid meshes for the simulation of physiological fluid-structure interaction problems based on data from medical images. Here we show how to simulate steady finite-Reynolds-number flow through a (rigid) iliac bifurcation. A particular feature of this problem is that, unlike the problem considered in `another tutorial,` the in- and outflow boundaries are not aligned with any coordinate planes. Parallel in- and outflow is therefore enforced by a Lagrange multiplier method, implemented using `oomph-lib's FaceElements`.

The problem studied here also serves as a "warm-up problem" for the `corresponding fluid-structure interaction problem` in which the vessel wall is elastic and deforms in response to the traction that the fluid exerts onto it.

We stress that the tutorial focuses on the implementation aspects, not the actual physics. Since the driver code discussed here is also used in the library's self-tests we deliberately use a very coarse mesh and restrict ourselves to steady flows. The results shown below are therefore unlikely to bear much resemblance to the actual flows that arise *in vivo*. The section How to make the simulation more realistic at the end of this tutorial provides several suggestions on how to make the simulation more realistic.

## 1.1 The problem (and results)

The two figures below show the geometry of the blood vessel (obtained from a scan of an iliac bifurcation, using the procedure discussed in `oomph-lib's VMTK tutorial`) and the flow field (velocity vectors and pressure contours) for a nominal Reynolds number of $Re = 10$. (See What does the Reynolds number mean in this problem? for a more detailed discussion of the Reynolds number.) The flow is driven by an applied pressure drop between the in- and outflow boundaries, as in the `previous example.`

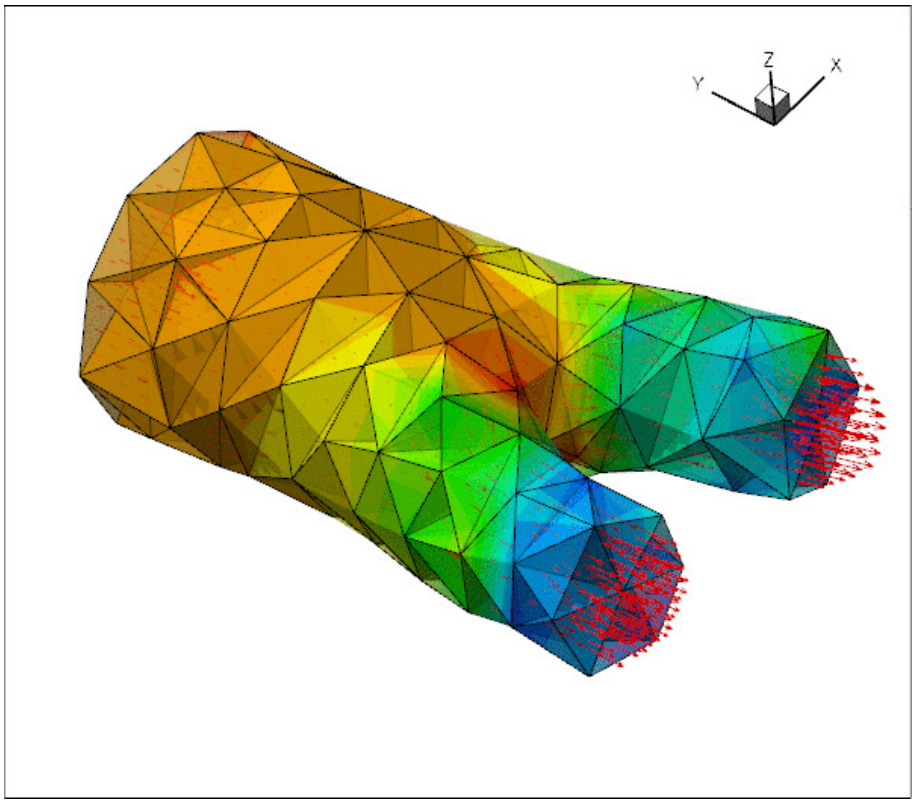


**Figure 1.1 Velocity field and pressure contours.**

The figure below shows more clearly that the in- and outflow from the upstream and downstream cross-sections is parallel, even though the cross-sections are not aligned with any the coordinate planes.

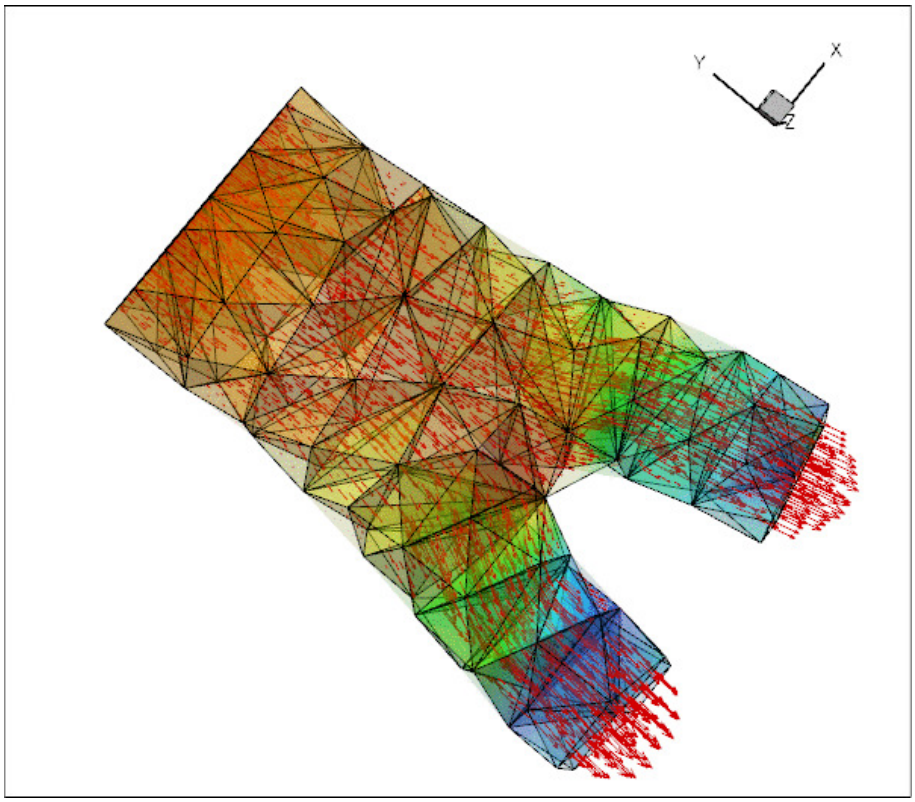


**Figure 1.2 Velocity field and pressure contours. Note the parallel in- and outflow.**

## 1.2 Imposing parallel in- and outflow

In most of the Navier-Stokes problems we have considered so far, the geometry of the fluid domain was such that the in- and outflow cross-sections were aligned with the Cartesian coordinate planes. In such geometries the imposition of parallel in- and outflow is straightforward as it only requires pinning of the transverse velocity components. A pressure drop between upstream and downstream ends can be applied by attaching `NavierStokesTractionElements` to the bulk elements that are adjacent to the relevant domain boundaries.
In the current problem, the in- and outflow cross-sections have some arbitrary orientation, implying that when the in- or outflow is parallel (or, more accurately, orthogonal to the in- or outflow cross sections), neither of the three velocity components vanishes. The easiest way to enforce parallel outflow in such situations is to employ 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. Physically, the Lagrange multipliers act as tangential tractions that enforce the parallel flow. The Lagrange multipliers introduce additional degrees of freedom into the problem and their implementation as `ImposeParallelOutflowElements` employs a technique similar to that used to enforce prescribed boundary displacements in solid mechanics problems. (This is discussed in another tutorial.) The `ImposeParallelOutflowElements` also allow the specification of a pressure that acts on the fluid (in the direction opposite to the outer unit normal on the bulk fluid element).

## 1.3 The driver code

The driver code is very similar to that discussed in another example where we imposed parallel outflow in coordinate-aligned in- and outflow cross-sections. We will therefore only discuss the changes required to impose parallel flow in cross-sections with arbitrary orientation.

### 1.3.1 Problem parameters

The namespace specifying the problem parameters contains the Reynolds number and the in- and outflow pressures (rather than the complete traction vector):

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

 /// Default Reynolds number
 double Re=10.0;

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


 /// Fluid pressure on outflow boundary
 double P_out=-0.5;
} //end namespace
```

### 1.3.2 The main function

The main function remains identical to that in the `problem with axis-aligned outflow.`

## 1.4 The Problem class

The `Problem` class is practically identical to that in the `problem with axis-aligned outflow`, apart from some trivial changes such as the provision of storage for meshes of `ImposeParallelOutflow↩ Elements` rather than `NavierStokesTractionElements`.

## 1.5 The Problem constructor

The problem constructor is also very similar. 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="fluid_iliac.1.node";
 string element_file_name="fluid_iliac.1.ele";
 string face_file_name="fluid_iliac.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);
```

Next, we set up a boundary lookup scheme that records 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. [The conversion code `create_fluid_and_solid_surface_mesh_from_fluid_xda_mesh.cc` lists the relation between the original boundary IDs and the new ones (obtained by giving each surface facet a separate boundary ID) at the end of the `*.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 '.poly' input file:

 // Fluid mesh inflow boundaries
 Inflow_boundary_id.resize(22);
 for(unsigned i=0; i<22; i++)
  {
   Inflow_boundary_id[i]=215+i;
  }
 // Fluid mesh outflow boundaries
 Outflow_boundary_id.resize(11);
 for(unsigned i=0; i<11; i++)
```

```
  {

   Outflow_boundary_id[i]=237+i;

  } // done outflow boundaries
```

We create the meshes containing the Lagrange multiplier elements and add all sub-meshes to the Problem's global mesh.

```
// Create meshes of Lagrange multiplier elements at inflow/outflow
//----------------------------------------------------------------

// Create the meshes
unsigned n=nfluid_traction_boundary();
Parallel_outflow_lagrange_multiplier_mesh_pt.resize(n);
for (unsigned i=0;i<n;i++)
 {
  Parallel_outflow_lagrange_multiplier_mesh_pt[i]=new Mesh;
 }
// Populate them with elements
create_parallel_outflow_lagrange_elements();

// 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(Parallel_outflow_lagrange_multiplier_mesh_pt[i]);
 }
// Build global mesh
build_global_mesh();
```

Next we apply the boundary conditions. We start by identifying the IDs of the boundaries that are subject to no-slip boundary conditions.

```
// Apply BCs
//----------
unsigned nbound=Fluid_mesh_pt->nboundary();

// Vector indicating the boundaries where we have no slip
std::vector<bool> pin_velocity(nbound, true);
// Loop over inflow/outflow boundaries
for (unsigned in_out=0;in_out<2;in_out++)
 {
  // Loop over in/outflow boundaries
  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];
     }

    pin_velocity[b]=false;
   }

 } // done identification of boundaries where velocities are pinned
```

Next we loop over all boundaries, visit their nodes and pin all three velocity components if the boundary is subject to a no-slip condition:

```
// Loop over all boundaries to apply no slip where required
for(unsigned b=0;b<nbound;b++)
 {
  if(pin_velocity[b])
   {
    unsigned num_nod=Fluid_mesh_pt->nboundary_node(b);
    for (unsigned inod=0;inod<num_nod;inod++)
     {
      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);
```

We now check if the node in question is also located on the in- and outflow boundaries...

```
       // Find out if the node is also located on an in- or outflow
       // boundary
       bool is_in_or_outflow_node=false;
       for (unsigned in_out=0;in_out<2;in_out++)
        {
         // Loop over boundaries with Lagrange multiplier elements
         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 bb=0;
           if (in_out==0)
            {
             bb=Inflow_boundary_id[i];
            }
           else
            {
             bb=Outflow_boundary_id[i];
            }

           if(nod_pt->is_on_boundary(bb))
            {
             is_in_or_outflow_node=true;
            }
          }
        } // now we know if it's on the an in- or outflow boundary...
```

...and if it is, we pin the Lagrange multipliers. They are stored after the values allocated by the "bulk" elements and we obtain the index of the first value associated with the Lagrange multipliers from the `BoundaryNodeBase::index_of_first_value_assigned_by_face_element()` function.

```
       // If its on an in- or outflow boundary pin the Lagrange multipliers
       if(is_in_or_outflow_node)
        {
         //Cast to a boundary node
         BoundaryNodeBase *bnod_pt =
          dynamic_cast<BoundaryNodeBase*>
          (Fluid_mesh_pt->boundary_node_pt(b,inod) );

         // What's the index of the first Lagrange multiplier
         // in the node's values?
         unsigned first_index=bnod_pt->index_of_first_value_assigned_by_face_element();

         // Pin the lagrange multiplier components
         // in the out/in_flow boundaries
         for (unsigned l=0;l<2;l++)
          {
           nod_pt->pin(first_index+l);
          }
        }
      }
    }
  } // end of BC
```

The rest of the constructor is unchanged. We pass the pointer to the Reynolds number to the elements and assign the equation numbers:

```
 // 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;

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

### 1.5.1 Creating the Lagrange multiplier elements

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

```
//============start_of_lagrange_multiplier_elements=====================
/// Create Lagrange multiplier elements that impose parallel outflow
//======================================================================
template<class ELEMENT>
void UnstructuredFluidProblem<ELEMENT>::
create_parallel_outflow_lagrange_elements()
{
 // Counter for number of Lagrange multiplier meshes
 unsigned count=0;

 // Loop over inflow/outflow boundaries
 for (unsigned in_out=0;in_out<2;in_out++)
  {
   // Loop over boundaries with Lagrange multiplier 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);

       // Build the corresponding lagrange element
       ImposeParallelOutflowElement<ELEMENT>* el_pt = new
        ImposeParallelOutflowElement<ELEMENT>(bulk_elem_pt,face_index);

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

       // Set the pointer to the prescribed pressure
       if (in_out==0)
        {
         el_pt->pressure_pt()= &Global_Parameters::P_in;
        }
       else
        {
         el_pt->pressure_pt()= &Global_Parameters::P_out;
        }
      }
     // Bump up counter
     count++;
    }
  }

}  // done
```

### 1.5.2 Post-processing

The post-processing routine is unchanged.

## 1.6 Comments and Exercises

### 1.6.1 What does the Reynolds number mean in this problem?

oomph-lib's implementation of the Navier-Stokes equations is based on their non-dimensional form so we typically expect the geometry of the problem to have been non-dimensionalised on a representative lengthscale, $\mathcal{L}$. When dealing with geometries that are obtained from medical images, the vessel coordinates are typically provided as dimensional quantities, e.g. in millimetres. Discarding the unit of the coordinates (i.e. using a non-dimensional coordinate $(x, y, z) = (1, 2, 3)$ to represent the point located at $(x^*, y^*, z^*) = (1\text{mm}, 2\text{mm}, 3\text{mm})$, say) is therefore equivalent to non-dimensionalising all lengths on a reference length of $\mathcal{L} = 1\text{mm}$. Recall that the Reynolds number is defined as

$$Re = \frac{\rho \mathcal{U} \mathcal{L}}{\mu}$$

where $\rho$ and $\mu$ are the fluid density and viscosity, respectively. The lengthscale $\mathcal{L} = 1\text{mm}$ does obviously not provide a measure of the dimension of our blood vessel whose diameter (at the inlet) is about $D \approx 16\text{mm}$. The nominal Reynolds number of $Re = 10$ used in the computations is therefore equivalent to an actual Reynolds number (formed with the vessel diameter) of

$$Re_D = \frac{\rho \mathcal{U} D}{\mu} = Re\, \frac{D}{\mathcal{L}} = 160,$$

where the velocity scale $\mathcal{U}$ is formed with the (dimensional) applied pressure drop $\Delta P^*$ between the in- and outflow cross-sections, as in the problem with axis-aligned in- and outflow cross-sections,

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

---

### 1.6.2 How to make the simulation more realistic

The simulation presented above is obviously very crude and serves primarily as a proof of concept. However, it is straightforward to address most of the shortcomings and we encourage you to explore the following improvements:

- Generate a finer fluid mesh using the instructions in oomph-lib's VMTK tutorial We are hoping to make the image files used in this tutorial available soon. Please contact us if you can't wait. Remember that you will have to update the enumeration of the domain boundaries if you change the mesh.

- Attach "flow extensions" to the in- and outflow cross-sections, using the technique described in oomph-lib's VMTK tutorial.

- Make the problem time-dependent and apply a period driving pressure drop.

---

## 1.7 Source files for this tutorial

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

demo_drivers/navier_stokes/vmtk_fluid/

- The driver code is:

demo_drivers/navier_stokes/vmtk_fluid/vmtk_fluid.cc

## 1.8 PDF file

A pdf version of this document is available.