codeStream – Highlights

codeStream – Boundary conditions

- There are many boundary conditions available in OpenFOAM®.
- But from time to time it may happen that you do not find what you are looking for.
- It is possible to implement your own boundary conditions, so in theory you can do whatever you want.
- Remember, you have the source code.
- To implement your own boundary conditions, you have three options:
    - Use **codeStream**.
    - Use high level programing.
    - Use an external library (*e.g.*, **swak4foam**).
- **codeStream** is the simplest way to implement boundary conditions, and most of the times you will be able to code boundary conditions with no problem.
- If you can not implement your boundary conditions using **codeStream**, you can use high level programming. However, this requires some knowledge on C++ and OpenFOAM® API.
- Hereafter, we are going to work with **codeStream** and basic high-level programming.
- We are not going to work with **swak4Foam** because it is an external library that is not officially supported by the OpenFOAM® foundation. However, it works very well and is relatively easy to use.

codeStream – Highlights

- Hereafter we will work with **codeStream**, which will let us program directly in the input dictionaries.
- With **codeStream**, we will implement our own boundary conditions and initial conditions without going thru the hustle and bustle of high-level programming.
- If you are interested in high level programming, refer to the supplements.
- In the supplemental slides, we address the following topics: building blocks, implementing boundary conditions using high level programming, modifying applications, implementing an application from scratch, and adding the scalar transport equation to icoFoam.
- High level programming requires some knowledge on C++ and OpenFOAM® API library. This is the hard part of programming in OpenFOAM®.
- Before doing high level programming, we highly recommend you to try with **codeStream**, most of the time it will work.
- Also, before modifying solvers or trying to implement your own solvers, understand the theory behind the FVM.
- Remember, you can access the API documentation in the following link, https://cpp.openfoam.org/v7

Implementing boundary conditions using codeStream

- OpenFOAM® includes the capability to compile, load and execute C++ code at run-time.
- This capability is supported via the directive **#codeStream**, that can be used in any input file for run-time compilation.
- This directive reads the entries **code** (compulsory), **codeInclude** (optional), **codeOptions** (optional), and **codeLibs** (optional), and uses them to generate the dynamic code.
- The source code and binaries are automatically generated and copied in the directory **`dynamicCode`** of the current case.
- The source code is compiled automatically at run-time.
- The use of **codeStream** is a very good alternative to avoid high level programming of boundary conditions or the use of external libraries.
- Hereafter we will use **codeStream** to implement new boundary conditions, but have in mind that **codeStream** can be used in any dictionary.

Implementing boundary conditions using codeStream

Body of the **codeStream** directive for boundary conditions

```
patch-name
{
    type            fixedValue;
    value           #codeStream
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Patch name

Use codeStream to set the value of the boundary condition

Files needed for compilation

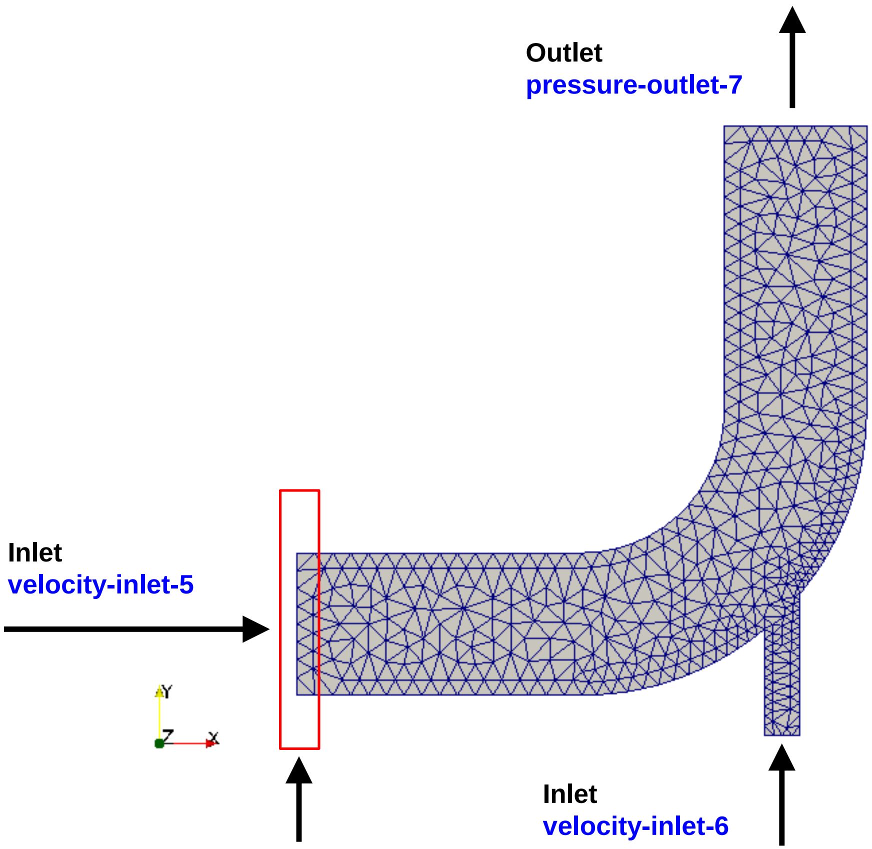
Compilation options

Libraries needed for compilation. Needed if you want to visualize the output of the boundary condition at time zero

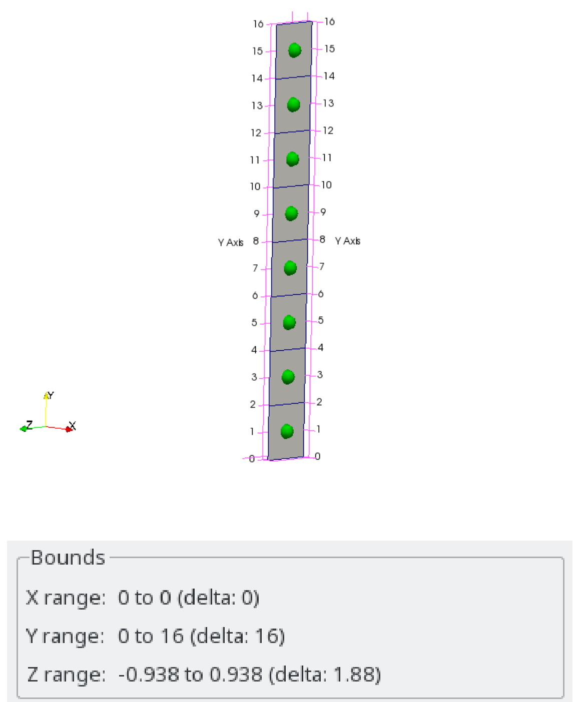
Insert your code here. At this point, you need to know how to access mesh information

Implementation of a parabolic inlet profile using **codeStream**

- Let us implement a parabolic inlet profile.
- The firs step is identifying the patch, its location and the dimensions.
- You can use paraview to get all visual references.



Parabolic inlet profile



Bounds of velocity-inlet-5 boundary patch

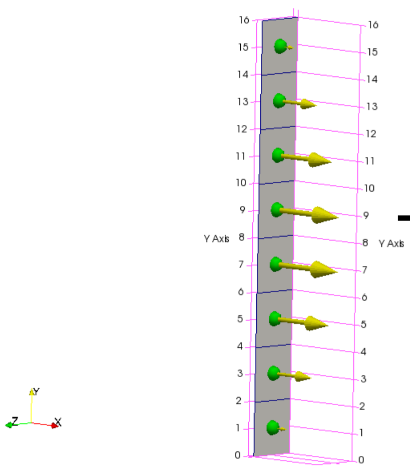
Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using **codeStream**

- We will use the following formula to implement the parabolic inlet profile

$$U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

- For this specific case *c* is the patch midpoint in the y direction (8), *r* is the patch semi-height or radius (8) and *Umax* is the maximum velocity.
- We should get a parabolic profile similar to this one,



$$U_{max}\left(1.0 - \frac{(y-8)^2}{64}\right)$$

Implementing boundary conditions using codeStream

- The **codeStream** BC in the body of the file *U* is as follows,

```
velocity-inlet-5
{
    type            fixedValue;
    value           #codeStream
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Patch name

Depending of what are you trying to do, you will need to add new files, options and libraries.

For most of the cases, this part is always the same.

Insert your code here.
At this point, you need to know how to access mesh information

Implementing boundary conditions using codeStream

- The **code** section of the **codeStream** BC in the body of the file *U* is as follows,

```
1    code
2    #{
3        const IOdictionary& d = static_cast<const IOdictionary&>
4        (
5            dict.parent().parent()
6         );
7
8         const fvMesh& mesh = refCast<const fvMesh>(d.db());
9         const label id = mesh.boundary().findPatchID("velocity-inlet-5");
10        const fvPatch& patch = mesh.boundary()[id];
11
12        vectorField U(patch.size(), vector(0, 0, 0));
13
14        ...
15        ...
16        ...
17    #};
```

To access boundary mesh information

Remember to update this value with the actual name of the patch

- Lines 3-11, are always standard, they are used to access boundary mesh information.
- In lines 3-6 we access the current dictionary.
- In line 8 we access the mesh database.
- In line 9 we get the label id (an integer) of the patch **velocity-inlet-5** (notice that you need to give the name of the patch).
- In line 10 using the label id of the patch, we access the boundary mesh information.
- In line 12 we initialize the vector field. The statement **patch.size()** gets the number of faces in the patch, and the statement **vector(0, 0, 0)** initializes a zero vector field in the patch.

Implementing boundary conditions using codeStream

- The **code** section of the **codeStream** BC in the body of the file U is as follows,

```
1     code
2     #{
3         ...
4         ...
5         ...
6         const scalar pi = constant::mathematical::pi;
7         const scalar U_0    = 2.;  //maximum velocity
8         const scalar p_ctr = 8.;  //patch center
9         const scalar p_r    = 8.;  //patch radius
10
11        forAll(U, i)          //equivalent to for (int i=0; patch.size()<i; i++)
12        {
13            const scalar y = patch.Cf()[i][1];
14            U[i] = vector(U_0*(1-(pow(y - p_ctr,2))/(p_r*p_r)), 0., 0.);
15        }
16
17        writeEntry(os, "", U);
18    #};
```

Index used to access the y coordinate

$0 \rightarrow x$

$1 \rightarrow y$

$2 \rightarrow z$

Assign input profile to vector field U (component x)

$$U_{max}\left(1.0 - \frac{(y-8)^2}{64}\right)$$

- In lines 6-17 we implement the new boundary condition.
- In lines 6-9 we declare a few constant needed in our implementation.
- In lines 11-15 we use a forAll loop to access the boundary patch face centers and to assign the velocity profile values. Notice the U was previously initialized.
- In line 13 we get the y coordinates of the patch faces center.
- In line 14 we assign the velocity value to the patch faces center.
- In line 17 we write the U values to the dictionary.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using **codeStream**

- This case is ready to run, the input files are located in the directory
$PTOFC/101programming/codeStream_BC/2Delbow_UparabolicInlet
- To run the case, type in the terminal,

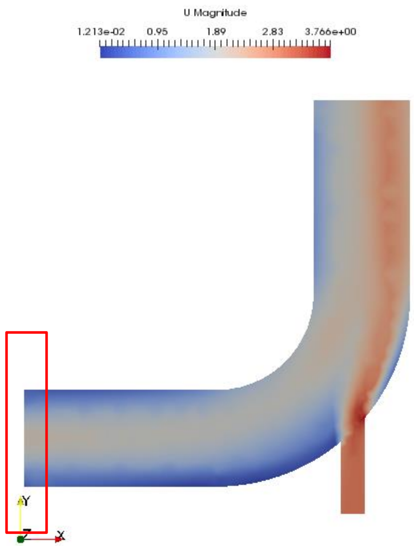
1. `$> cd $PTOFC/101programming/codeStream_BC/2Delbow_UparabolicInlet`
2. `$> foamCleanTutorials`
3. `$> fluentMeshToFoam ../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh`
4. `$> icoFoam | tee log`
5. `$> paraFoam`

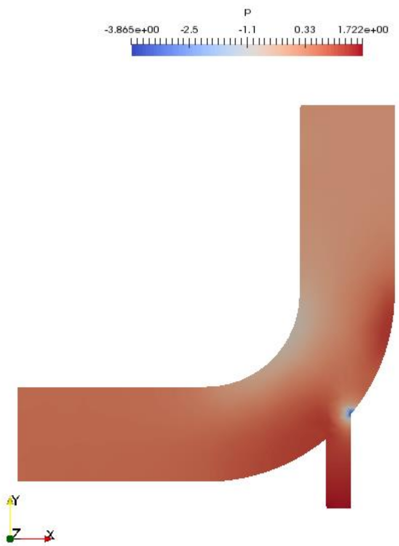
- The **codeStream** boundary condition is implemented in the file *0/U*.

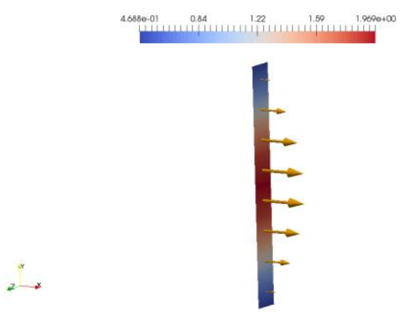
Implementing boundary conditions using codeStream

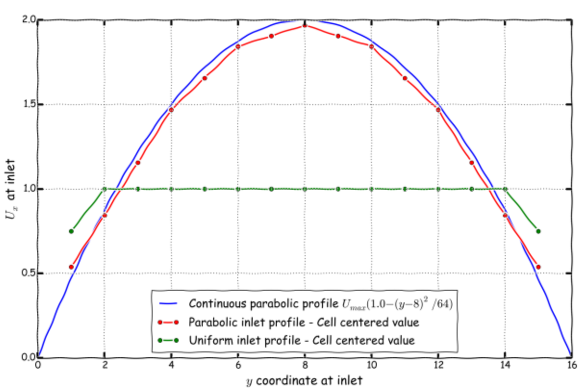
Implementation of a parabolic inlet profile using **codeStream**

- If everything went fine, you should get something like this









Implementing boundary conditions using codeStream

codeStream works with scalar and vector fields

- We just implemented the input parabolic profile using a vector field.
- You can do the same using a scalar field, just proceed in a similar way.
- Remember, now we need to use scalars instead of vectors.
- And you will also use an input dictionary holding a scalar field.

```
code
#{
    ...
    ...
    ...
    scalarField S(patch.size(), scalar(0) );

    forAll(S, i)
    {
        const scalar y = patch.Cf()[i][1];
        S[i] = scalar( 2.0*sin(3.14159*y/8.) );
    }

    writeEntry(os, "", S);
#};
```

Initialize scalar field

Loop using scalar field size

Write profile values in scalar field

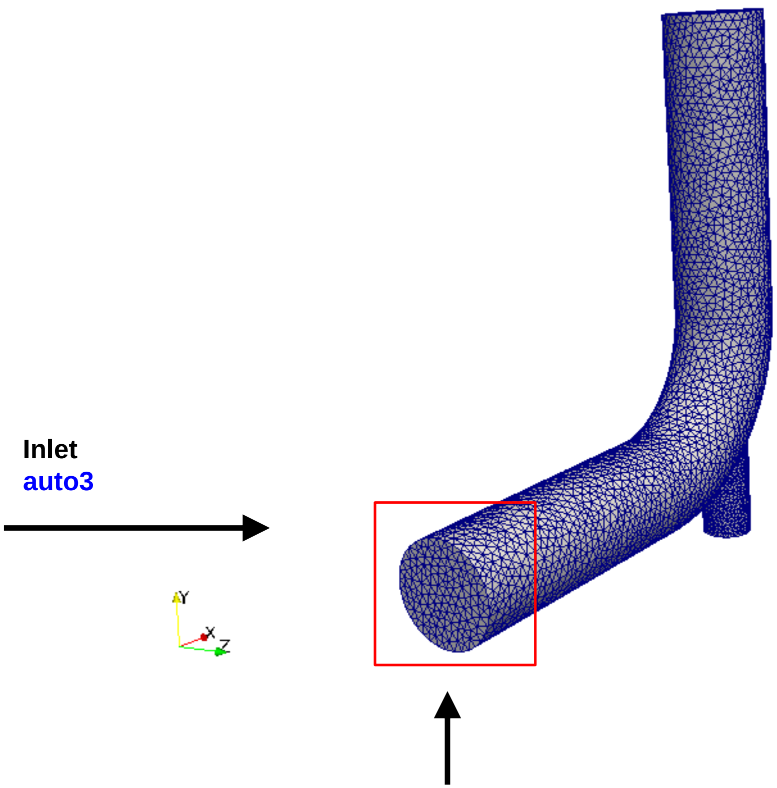
Write output to input dictionary

Notice that the name of the field does not need to be the same as the name of the input dictionary

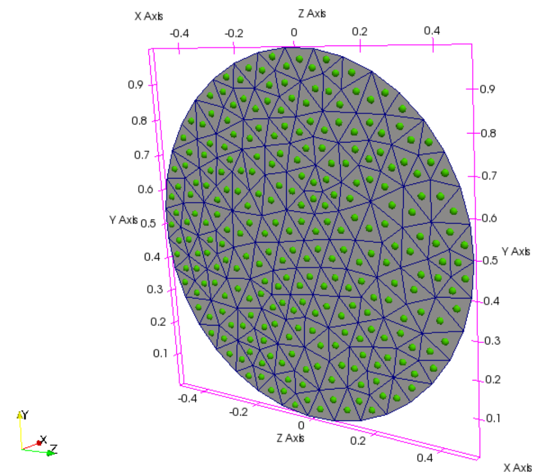
Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using **codeStream**

- Let us work in a case a little bit more complicated, a paraboloid input profile.
- As usual, the first step is to get all the spatial references.



Paraboloid inlet profile



Bounds

X range: 0 to 0 (delta: 0)

Y range: 0.000503 to 0.999 (delta: 0.999)

Z range: -0.498 to 0.5 (delta: 0.998)

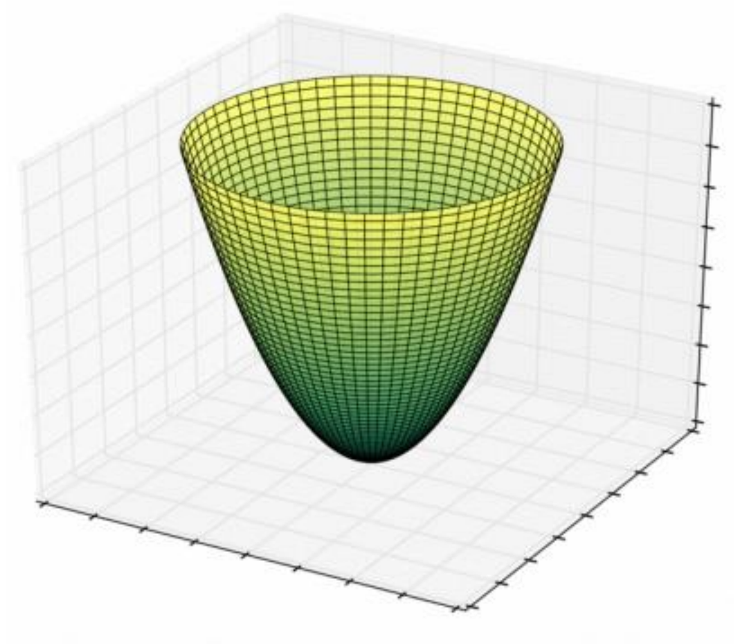
Bounds of auto3 boundary patch

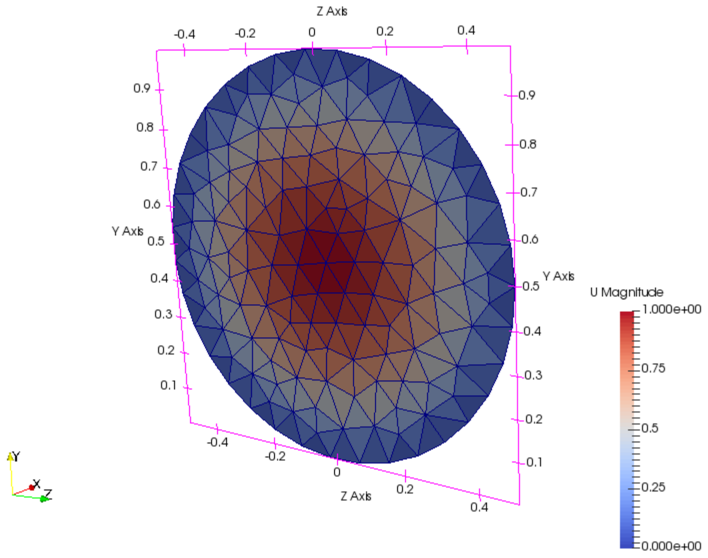
Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using **codeStream**

- We will implement the following equation in the boundary patch **auto3**.

$$U = \left(\frac{z}{0.5}\right)^2 + \left(\frac{y-0.5}{0.5}\right)^2 - 1$$





Implementing boundary conditions using codeStream

- The **codeStream** BC in the body of the file *U* is as follows,

```
auto3
{
    type            fixedValue;
    value           #codeStream
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Patch name

For most of the cases, this part is always the same. But depending of what are you trying to do, you will need to add more files, options and libraries.

Insert your code here.
We will implement the following equation

$$U = \left(\frac{z}{0.5}\right)^2 + \left(\frac{y-0.5}{0.5}\right)^2 - 1$$

Implementing boundary conditions using codeStream

- Hereafter, we only show the actual implementation of the **codeStream** boundary condition.
- The rest of the body is a template that you can always reuse. Including the section of how to access the dictionary and mesh information.
- Remember, is you are working with a vector, you need to use vector fields. Whereas, if you are working with scalars, you need to use scalars fields.

```
code
#{
    ...
    ...
    ...
    vectorField U(patch.size(), vector(0, 0, 0) );

    const scalar s  = 0.5;

    forAll(U, i)
    {
        const scalar x = patch.Cf()[i][0];
        const scalar y = patch.Cf()[i][1];
        const scalar z = patch.Cf()[i][2];

        U[i] = vector(-1.*(pow(z/s, 2) + pow((y-s)/s,2) - 1.0), 0, 0);
    }

    writeEntry(os, "", U);
#};
```

Initialize vector field

Initialize scalar

Access faces center coordinates (x, y, and z)

$$U = \left(\frac{z}{0.5}\right)^2 + \left(\frac{y-0.5}{0.5}\right)^2 - 1$$

Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using **codeStream**

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_BC/3Delbow_Uparaboloid/`**
- To run the case, type in the terminal,

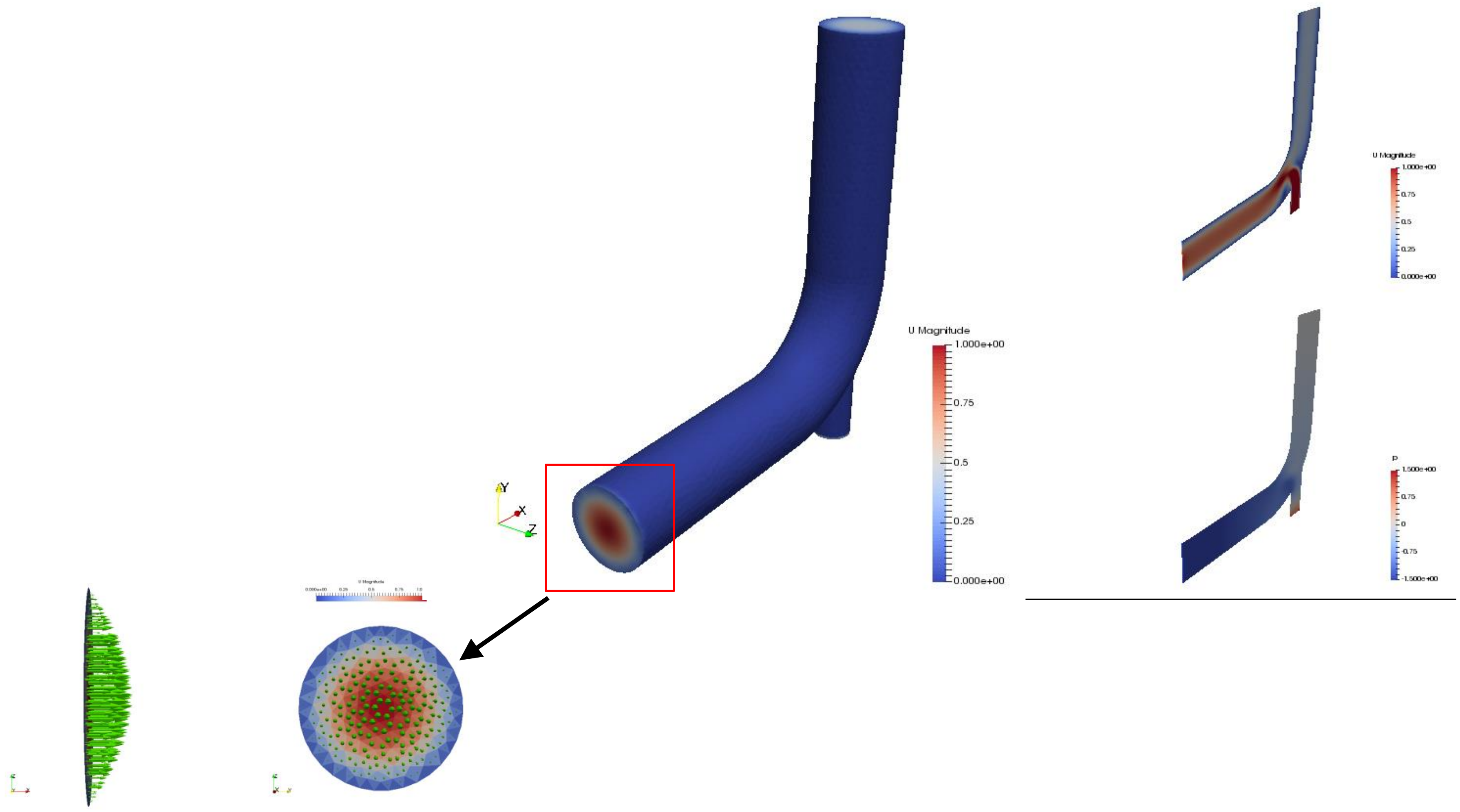
```
1. $> cd $PTOFC/101programming/codeStream_BC/3Delbow_Uparaboloid/
2. $> foamCleanTutorials
3. $> gmshToFoam ../../../meshes_and_geometries/gmsh_elbow3d/geo.msh
4. $> autoPatch 75 -overwrite
5. $> createPatch -overwrite
6. $> renumberMesh -overwrite
7. $> icoFoam | tee log
8. $> paraFoam
```

- The **codeStream** boundary condition is implemented in the file *`0/U`*.

Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using **codeStream**

- If everything went fine, you should get something like this



Implementing boundary conditions using codeStream

codedFixedValue and **codedMixed** boundary conditions

- OpenFOAM® also includes the boundary conditions **codedFixedValue** and **codedMixed**.
- These boundary conditions are derived from **codeStream** and work in a similar way.
- They use a friendlier notation and let you access more information of the simulation database (e.g. time).
- The source code and binaries are automatically generated and copied in the directory **`dynamicCode`** of the current case.
- Another feature of these boundary conditions, is that the **code** section can be read from an external dictionary (*`system/codeDict`*), which is run-time modifiable.
- The boundary condition **codedMixed** works in similar way. This boundary condition gives you access to fixed values (Dirichlet BC) and gradients (Neumann BC).
- Let us implement the parabolic profile using **codedFixedValue**.

Implementing boundary conditions using codeStream

Body of the **codedFixedValue** boundary conditions

```
patch-name
{
    type            codedFixedValue;
    value           uniform (0 0 0);


    redirectType    name_of_BC;


    codeOptions
    #{
        -I$(LIB_SRC)/finiteVolume/lnInclude \
        -I$(LIB_SRC)/meshTools/lnInclude
    #};

    codeInclude
    #{
        #include "fvCFD.H"
        #include <cmath>
        #include <iostream>
    #};

    code
    #{

    #};
}
```

Patch name

Use codedFixedValue and initializations

Unique name of the new boundary condition.
If you have more codedFixedValue BC, the names must be different

Compilation options

Files needed for compilation

In this section we do the actual implementation of the boundary condition.
This is the only part of the body that you will need to change. The rest of the body is a template that you can always reuse.

Implementing boundary conditions using codeStream

- The **code** section of the **codeStream** BC in the body of the file U is as follows,

```
1     code
2     #{
3         const fvPatch& boundaryPatch = patch();
4         const vectorField& Cf = boundaryPatch.Cf();
5         vectorField& field = *this;
6
7         scalar U_0 = 2, p_ctr = 8, p_r = 8;
8
9         forAll(Cf, faceI)
10        {
11            field[faceI] = vector(U_0*(1-(pow(Cf[faceI].y()-p_ctr,2))/(p_r*p_r)),0,0);
12        }
13     #};
```

$$U_{max}\left(1.0 - \frac{(y-8)^2}{64}\right)$$

- Lines 3-5, are always standard, they give us access to mesh and field information in the patch.
- The coordinates of the faces center are stored in the vector field **Cf** (line 4).
- In this case, as we are going to implement a vector profile, we initialize a vector field where we are going to assign the profile (line 5).
- In line 7 we initialize a few constants that will be used in our implementation.
- In lines 9-12 we use a forAll loop to access the boundary patch face centers and to assign the velocity profile values.
- In line 11 we do the actual implementation of the boundary profile (similar to the **codeStream** case). The vector field was initialize in line 5.

Implementing boundary conditions using codeStream

codedFixedValue and **codedMixed** boundary conditions

- As you can see, the syntax and use of the **codedFixedValue** and **codedMixed** boundary conditions is much simpler than **codeStream**.
- You can use these instructions as a template. At the end of the day, you only need to modify the **code** section.
- Depending of what you want to do, you might need to add new headers and compilation options.
- Remember, is you are working with a vector, you need to use vector fields. Whereas, if you are working with scalars, you need to use scalars fields.
- One disadvantage of these boundary conditions, is that you can not visualize the fields at time zero. You will need to run the simulation for at least one iteration.
- On the positive side, accessing time and other values from the simulation database is straightforward.
- Time can be accessed by adding the following statement,

```
this->db().time().value()
```

Implementing boundary conditions using codeStream

- Let us add time dependency to the parabolic profile.

```
1     code
2     #{
3         const fvPatch& boundaryPatch = patch();
4         const vectorField& Cf = boundaryPatch.Cf();
5         vectorField& field = *this;
6
7         scalar U_0 = 2, p_ctr = 8, p_r = 8;
8
9         scalar t = this->db().time().value();
10
11        forAll(Cf, faceI)
12        {
13           field[faceI] = vector(sin(t)*U_0*(1-(pow(Cf[faceI].y()-p_ctr,2))/(p_r*p_r)),0,0);
14        }
15     #};
```

Time

Time dependency

$$sin(t) \times U_{max} \left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

- This implementation is similar to the previous one, we will only address how to deal with time.
- In line 8 we access simulation time.
- In line 13 we do the actual implementation of the boundary profile (similar to the **codeStream** case). The vector field was initialize in line 5 and time is accessed in line 9.
- In this case, we added time dependency by simple multiplying the parabolic profile by the function **sin(t)**.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using **codedFixedValue**

- This case is ready to run, the input files are located in the directory
  `$PTOFC/101programming/codeStream_BC/2Delbow_UparabolicInlet_timeDep`
- To run the case, type in the terminal,

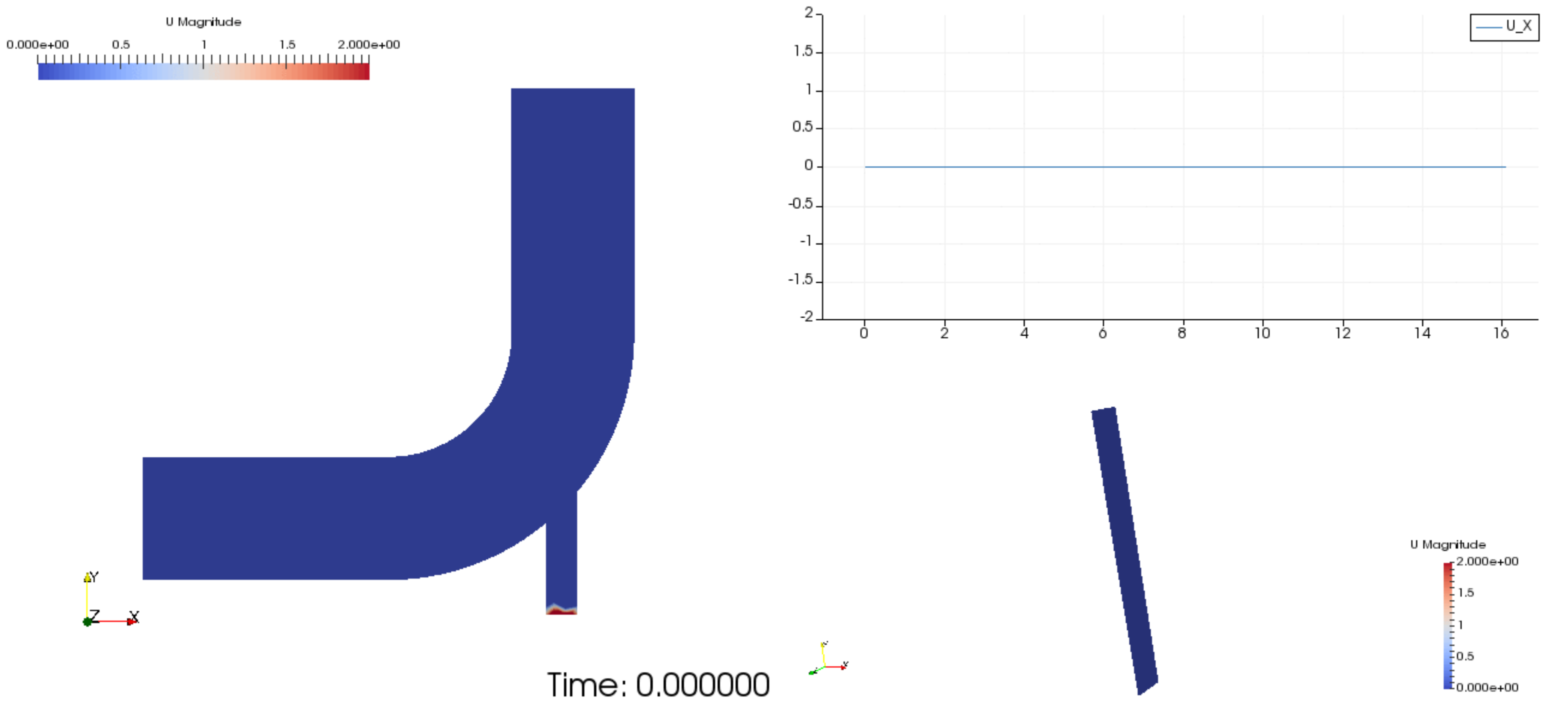
```
1. $> cd $PTOFC/101programming/codeStream_BC/2Delbow_UparabolicInlet_timeDep
2. $> foamCleanTutorials
3. $> fluentMeshToFoam ../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh
4. $> icoFoam | tee log
5. $> paraFoam
```

- The **codeStream** boundary condition is implemented in the file *0/U*.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using **codedFixedValue**

- If everything went fine, you should get something like this



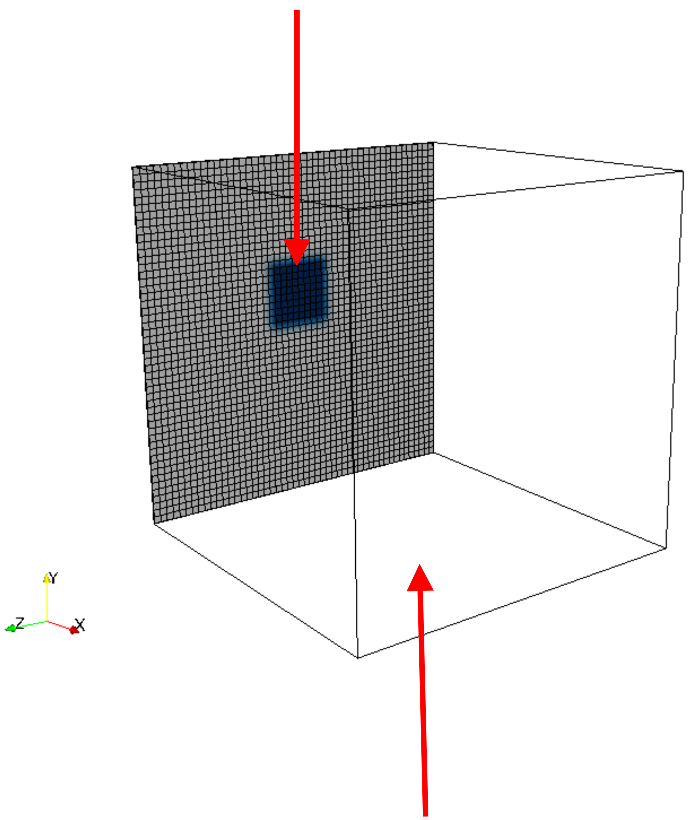
www.wolfdynamics.com/wiki/BCIC/elbow_unsBC1.gif

Implementing boundary conditions using codeStream

Filling a tank using **codedFixedValue**

- Let us do a final example.
- We will deal with scalar and vector fields at the same time.
- We will use **codedFixedValue**.
- For simplicity, we will only show the **code** section of the input files.
- Remember, the rest of the body can be used as a template.
- And depending of what you want to do, you might need to add new headers, libraries, and compilation options.
- Hereafter we will setup an inlet boundary condition in a portion of an existing patch.
- By using **codedFixedValue** BC, we do not need to modify the actual mesh topology.
- We will assign a velocity field and a scalar field to a set of faces (dark area in the figure).
- We are going to simulate filling a tank with water.
- We will use the solver **interFoam**.

Water enters here
This is a face selection in a single boundary patch



The tank is initially empty

Implementing boundary conditions using codeStream

- Definition of the vector field boundary condition (dictionary file *U*),

Name of the patch where we want to implement the boundary condition

```
leftWall
{
  type            codedFixedValue;
  value           uniform (0 0 0);
  redirectType    inletProfile1;

  code
  #{
     const fvPatch& boundaryPatch = patch();
     const vectorField& Cf = boundaryPatch.Cf();
     vectorField& field = *this;

     scalar minz = 0.4;
     scalar maxz = 0.6;
     scalar miny = 0.5;
     scalar maxy = 0.7;

     scalar t = this->db().time().value();
     ...
     ...
     ...
   #};
 }
```

Use codedFixedValue BC and initialize value. The initialization is only needed for paraview in order to visualize something at time zero.

Unique name of the BC
Do not use the same name in other patches

Access boundary mesh information and initialize vector field **field**

Initialize variables

Access time

Implementing boundary conditions using codeStream

- Definition of the vector field boundary condition (dictionary file *U*),

```
7     code
8     #{
          ...
          ...
          ...
19
20        forAll(Cf, faceI)
21        {
22
23            if (
24                 (Cf[faceI].z() > minz) &&
25                 (Cf[faceI].z() < maxz) &&
26                 (Cf[faceI].y() > miny) &&
27                 (Cf[faceI].y() < maxy)
28               )
29               {
30                   if ( t < 1.)
31                   {
32                       field[faceI] = vector(1,0,0);
33                   }
34                   else
35                   {
36                       field[faceI] = vector(0,0,0);
37                   }
38               }
39        }
40      #};
41      }
```

Code section. The actual implementation of the BC is done in this section

Loop using size of boundary patch (**Cf**) and iterator **faceI**.
This is equivalent to:
    for (int faceI=0; Cf.size()<faceI; faceI++)

Use conditional structure to select faces according to the variables defined in lines 13-16

Use conditional structure to add time dependency and assign values to the selected faces.
The variable **field** was initialize in line 11.

Implementing boundary conditions using codeStream

- Definition of the scalar field boundary condition (dictionary file *alpha.water*),

Name of the patch where we want to implement the boundary condition

```
leftWall
{
    type            codedFixedValue;
    value           uniform 0;
    redirectType    inletProfile2;

    code
    #{
        const fvPatch& boundaryPatch = patch();
        const vectorField& Cf = boundaryPatch.Cf();
        scalarField& field = *this;

        field = patchInternalField();

        scalar minz = 0.4;
        scalar maxz = 0.6;
        scalar miny = 0.5;
        scalar maxy = 0.7;

        scalar t = this->db().time().value();

        ...
        ...
        ...
     #};
 }
```

Use codedFixedValue BC and initialize value. The initialization is only needed for paraview in order to visualize something at time zero.

Unique name of the BC
Do not use the same name in other patches

Code section. The actual implementation of the BC is done in this section

Access boundary mesh information and initialize scalar field **field**

Assign value from the internal field to the patch

Initialize variables

Access time

Implementing boundary conditions using codeStream

- Definition of the scalar field boundary condition (dictionary file `alpha.water`),

```
7        code
8        #{
             ...
             ...
             ...
22
23           forAll(Cf, faceI)
24           {
25               if (
26                      (Cf[faceI].z() > minz) &&
27                      (Cf[faceI].z() < maxz) &&
28                      (Cf[faceI].y() > miny) &&
29                      (Cf[faceI].y() < maxy)
30                  )
31                  {
32                      if ( t < 1.)
33                      {
34                          field[faceI] = 1.;
35                      }
36                      else
37                      {
38                          field[faceI] = 0.;
39                      }
40                  }
41           }
42        #};
43    }
```

Code section. The actual implementation of the BC is done in this section

Loop using size of boundary patch (**Cf**) and iterator **faceI**.
This is equivalent to:
for (int faceI=0; Cf.size()<faceI; faceI++)

Use conditional structure to select faces according to the variables defined in lines 13-16

Use conditional structure to add time dependency and assign values to the selected faces. The variable **field** was initialize in line 11.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using **codedFixedValue**

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_BC/fillBox_BC/`**
- To run the case, type in the terminal,

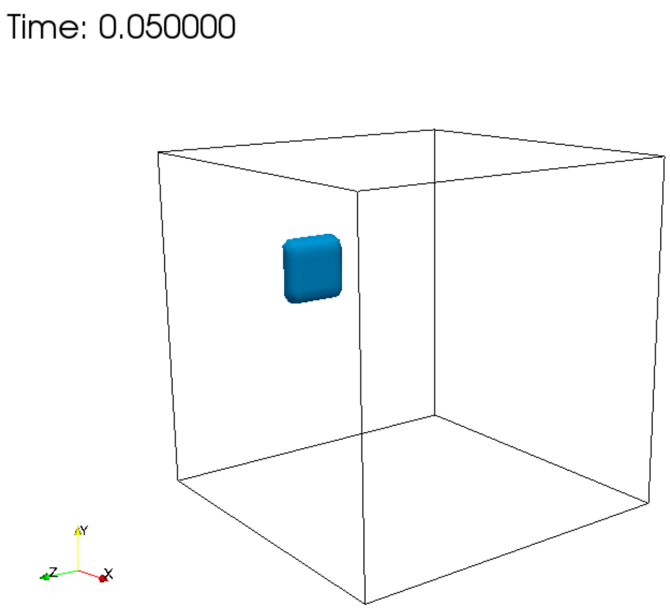
```
1. $> cd $PTOFC/101programming/codeStream_BC/fillBox_BC/
2. $> foamCleanTutorials
3. $> blockMesh
4. $> decomposePar
5. $> mpirun -np 4 interFoam -parallel | tee log
6. $> reconstructPar
7. $> paraFoam
```

- As you can see, we can also run in parallel with no problem.
- FYI, the stand alone version of Paraview does not handle **codedFixedValue** BC.
- To visualize the results, you need to use `paraFoam` with no options (avoid the option `–builtin`).

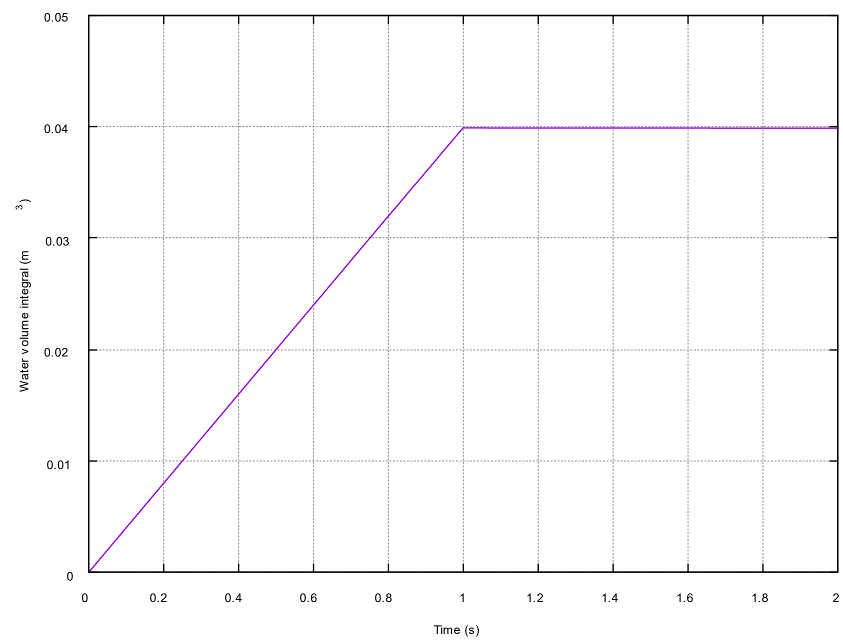
Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using **codedFixedValue**

- If everything went fine, you should get something like this



Visualization of water phase (alpha.water)



Volume integral of water entering the domain

www.wolfdynamics.com/wiki/BCIC/filltank1.gif