CFD with OpenSource software, assignment 3

Tutorial perforatedPlateBoundary

Developed for OpenFOAM-1.7.x

Author:
Mohammad Irannezhad

Peer reviewed by:
Daniel Grönberg
Jelena Andric

November 2, 2010

1 Introduction

This tutorial describes how to program a new boundary condition that takes care of a perforated plate as a boundary. It also includes the instructions on how to declare it in the boundary conditions list. The pitzDaily test case in 3d is used to validate the implementation of the new boundary condition. A perforated plate is a plate with several holes in it which is usually used at the flow inlet. Figure 1.1 shows a perforated plate with a set of swirler vanes around it used as the inlet in a typical low swirl burner. The perforated plate splits the inlet flow into several jets. The goal is to set up a new kind of boundary condition that can handle such inlet conditions where some part of the inlet is wall and some part of it is an inlet.

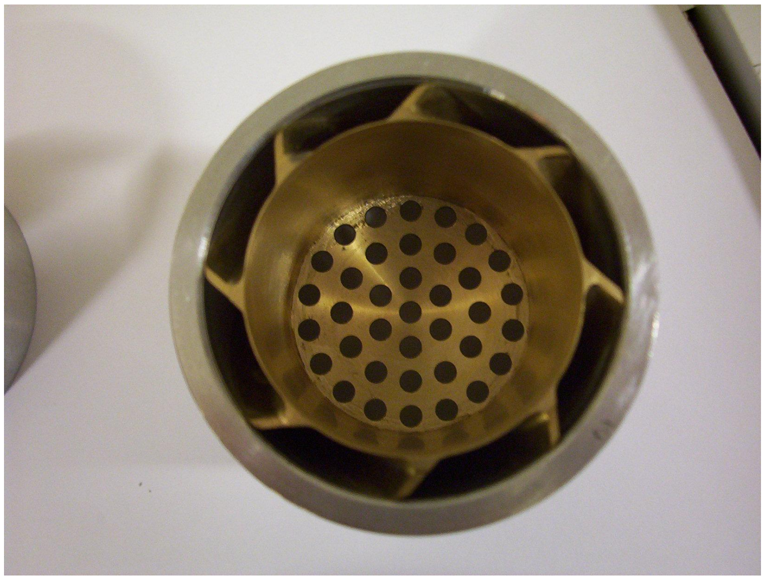

Figure 1: Perforated plate.

2 Implementation alternatives

There are several strategies that can be considered when setting up such a boundary condition depending on how you want to treat the wall. In general, OpenFOAM considers two kinds of boundary conditions:

- Boundary conditions that have some geometrical constraints applied to it. These include empty, symmetryPlane, etc.
- Boundary conditions without any physical constraints that are all of type *patch*.

A wall does not have any geometrical constraint and can be simply treated as a *patch* with zero pressure gradient. However, OpenFOAM recognizes between a simple *patch* and a *wall type* patch. The only reason to separate *wall type* from *patch type* is to be able to apply turbulent wall treatment on it, e.g. wall functions.
The first strategy is to consider the wall part of the perforated plate as a normal *patch type* but specify different values to the hole/wall parts of the plate. It is important to explicitly specify zero pressure gradient condition to the wall part.
The second strategy is to split the hole/wall part of the plate into real *patch type* and *wall type* boundaries. This necessitate the manipulation of the mesh.
Detailed implementations and pros and cons of the two alternatives will be discussed.

3 Alternative 1: as boundary condition

3.1 Getting started

Start as usual by finding some boundary condition that does almost what you want and copy it to the working directory and change the name of all the files and directories. Here this boundary condition is used with by *simpleFoam* solver as a static library.

```
cd $FOAM_USER_APP
cp -r $FOAM_SOLVERS/incompressible/simpleFoam .
mv simpleFoam simpleFoamPP
cd simpleFoamPP
mv simpleFoam.C simpleFoamPP.C
wclean
cp $FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/oscillatingFixedValue/*.* .
mv oscillatingFixedValueFvPatchField.H      perforatedPlateFixedValueFvPatchField.H
mv oscillatingFixedValueFvPatchField.C      perforatedPlateFixedValueFvPatchField.C
mv oscillatingFixedValueFvPatchFields.H     perforatedPlateFixedValueFvPatchFields.H
mv oscillatingFixedValueFvPatchField.C      perforatedPlateFixedValueFvPatchFields.C
mv oscillatingFixedValueFvPatchFieldFwd.H   perforatedPlateFixedValueFvPatchFieldFwd.H
```

Then using any text editor we change all the occurances of **oscillating** to **perforatedPlate** in all the files whose names start with **perforatedPlate**.

3.2 Modifications

Start with **perforatedPlateFixedValueFvPatchField.H** file and do the following modifications:

- Replace the private data with the following private data,

```
            //- Value of the field at the holes
            Field<Type> PPHolesValue_;

            //- Value of the field at the wall
```

```
        Field<Type> PPWallValue_;

        //- Centers of the holes
        List<vector> PPHolesCenters_;

        //- Radii of the holes
        List<scalar> PPHolesRadii_;

        //Face centers
        Field<vector> faceCenters_;

        //- Current time index
        label curTimeIndex_;
```

- Comment or delete the declaration of the *currentScale()* member function.

```
//    scalar currentScale() const;
```

- Replace the private member functions with the following

```
            //-
            const Field<Type>& PPHolesValue() const
            {
                return PPHolesValue_;
            }
            //-
            Field<Type>& PPHolesValue()
            {
                return PPHolesValue_;
            }
            //-
            const Field<Type>& PPWallValue() const
            {
                return PPWallValue_;
            }
            //-
            Field<Type>& PPWallValue()
            {
                return PPWallValue_;
            }
            //-
            List<vector> PPHolesCenters() const
            {
                return PPHolesCenters_;
            }
            List<vector>& PPHolesCenters()
            {
                return PPHolesCenters_;
            }
            //-
            List<scalar> PPHolesRadii() const
            {
                return PPHolesRadii_;
            }
            List<scalar>& PPHolesRadii()
            {
                return PPHolesRadii_;
            }
            //-
            Field<vector> faceCenters() const
            {
                return faceCenters_;
            }
            Field<vector>& faceCenters()
            {
                return faceCenters_;
            }
```

In the file **perforatedPlateFixedValueFvPatchField.C** do the following changes:

- Comment or delete the definition of the *currentScale()* function.
- Modify the constructors as follows.

```
template<class Type>
perforatedPlateFixedValueFvPatchField<Type>::perforatedPlateFixedValueFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchField<Type>(p, iF),
    PPHolesValue_("PPHolesValue", dict, p.size()),
    PPWallValue_("PPWallValue", dict, p.size()),
    PPHolesCenters_(dict.lookup("PPHolesCenters")),
    PPHolesRadii_(dict.lookup("PPHolesRadii")),
    faceCenters_(p.Cf())

{
  if (dict.found("value"))
   {
     fixedValueFvPatchField<Type>::operator==
      (
          Field<Type>("value", dict, p.size())
      );
   }
   else
   {

     fvPatchField<Type>::operator==(PPHolesValue_);
      Field<Type>& patchfield = *this;
      forAll(p.Cf(),i)  //go through all the cells in the current patch
{
          bool isHole=false;
  forAll(PPHolesRadii_,j)//go through all the holes
  {
    if(mag(p.Cf()[i]-PPHolesCenters_[j])<=PPHolesRadii_[j]){isHole=true;}//find if the face is in
          }
  if(!isHole){patchfield.data()[i]= PPWallValue_[i];}
        }
   }
}
```

- Modify all the constructors according to the new private data.
- Modify *rmap* and *autoMap* functions as follows.

```
template<class Type>
void perforatedPlateFixedValueFvPatchField<Type>::autoMap
(
    const fvPatchFieldMapper& m
)
{
    fixedValueFvPatchField<Type>::autoMap(m);
    PPHolesValue_.autoMap(m);
    PPWallValue_.autoMap(m);
    faceCenters_.autoMap(m);
}

template<class Type>
void perforatedPlateFixedValueFvPatchField<Type>::rmap
(
    const fvPatchField<Type>& ptf,
    const labelList& addr
)
{
    fixedValueFvPatchField<Type>::rmap(ptf, addr);

    const perforatedPlateFixedValueFvPatchField<Type>& tiptf =
        refCast<const perforatedPlateFixedValueFvPatchField<Type> >(ptf);

    PPHolesValue_.rmap(tiptf.PPHolesValue_, addr);
    PPWallValue_.rmap(tiptf.PPWallValue_, addr);
    faceCenters_.rmap(tiptf.faceCenters_, addr);
}
```

- Modify the *updateCoeffs()* member function according to

```
  template<class Type>
void perforatedPlateFixedValueFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
      Info<<"updated"<<endl;
     return;

    }

    if (curTimeIndex_ != this->db().time().timeIndex())
    {
        Field<Type>& patchfield = *this;
        patchfield = PPHolesValue_;

forAll(this->faceCenters(),i)//go through all the faces of the current patch
{
         bool isHole=false;
  forAll(PPHolesRadii_,j)//go through all the holes
  {
    if(mag(this->faceCenters()[i]-PPHolesCenters_[j])<=PPHolesRadii_[j]){isHole=true;}
         }
  if(!isHole){patchfield.data()[i]= PPWallValue_[i];}
        }
        curTimeIndex_ = this->db().time().timeIndex();

    }

    fixedValueFvPatchField<Type>::updateCoeffs();
    }
```

- Modify the *write(os)* function.

```
template<class Type>
void perforatedPlateFixedValueFvPatchField<Type>::write(Ostream& os) const
{
    fixedValueFvPatchField<Type>::write(os);
    PPHolesValue_.writeEntry("PPHolesValue", os);
    PPWallValue_.writeEntry("PPWallValue", os);
    os.writeKeyword("PPHolesCenters")
        << PPHolesCenters_ << token::END_STATEMENT << nl;
    os.writeKeyword("PPHolesRadii")
        << PPHolesRadii_ << token::END_STATEMENT << nl;

}
```

There are a couple of points about the above modifications:

- The private data *faceCenter*\ is initialized as the face centers of the current *fvPatch* object.
- The *updateCoeffs()* is a virtual function and is originally defined to take no input arguments. It is therfore called with no arguments, thus in the definition of this function we have no access to the face centers of the current *fvPatch* object. This is the reason behind having the *faceCenter()* as a member function.

3.3 Compiling

To have this boundary condition as a static boundary you need to include it in the new *simpleFoamPP* solver so the following line is added to the header of the *simpleFoamPP.C*

```
#include "perforatedPlateFixedValueFvPatchField.H"
```

And the **Make/files** is modified to assure including this new bounday condition, to change the name of the executable and save it in the correct location. Then compile with *wmake*.

```
simpleFoamPP.C
perforatedPlateFixedValueFvPatchFields.C

EXE = $(FOAM_USER_APPBIN)/simpleFoamPP
```

3.4 Specify the boundary condition

The perforated plate is first declared in the **blockMeshDict** as either a *patch* or a *wall* and then the new boundary condition is specified in the fields in **0/** directory as follows (example for velocity field **U**):

```
      {
        type            perforatedPlateFixedValue;
value           uniform (0 0 0);
        PPHolesValue    uniform (10 0 0);
        PPWallValue     uniform (0 0 0);
        PPHolesCenters  ((-0.0206 0.008 0)(-0.0206 0.016 0)(-0.0206 0.022 0));
        PPHolesRadii    (0.004 0.002 0.002);
      }
```

3.5 Pros And Cons

There are a number of advantages and disadvantages with this approach. Advantages are

- It is very easy and straight forward to do it.
- We are at the bottom of the class hierarchy of *OpenFOAM* and we can not possibly affect any other class.
- It can be used for most of the real world problems where a perforated plate is used at the inlet where flow is normal to the plate. This will be discussed more in disadvantages below.

Disadvantages are:

- For any new derived type of *fvPatchFiled* a new boundary condition should be set up. It means that if you want an oscillating value on the boundary you should program *perforated-PlateOscillatingFixedValue* boundary condition.
- It is not possible (or very difficult if possible) to have different basic *fvPatchFiled*s on wall and hole parts of the perforated plate. For instance it is not possible to have *zero Gradient* for the wall part and *fixed Value* on the hole part.
- The whole perforated plate is set to the same *type* (*patch* or *wall*) so when it is needed to really distinguish between a *wall* and a *patch* ,e.g using wall functions, it fails. This may happen when the flow is tangential to the plate.

4 Alternative 2: Manipulation of the mesh

Separating the faces of the perforated plate boundary into two separated *fvPatchField*s can be done in two ways

- Generate the mesh with one single *patch* for the whole perforated plate and then manipulate the mesh by *polyTopoChanger* functions.
- Step into the mesh generating utility *blockMesh* and change the mesh topology directly there.

The second option is used and described in this document.

4.1 How the mesh is set up?

The *blockMesh* utility reads and uses a dictionary *blockMeshDict* to construct a *polyMesh* class object which is then written to some files which are then read by the solvers. The mesh is saved as the following files:

- **points** which contains a list of vectors showing the location of the *vertices*.
- **faces** which contains a list of lists where each list contains the index of *vertices* constructing the *face*.
- **owner** which contains a list of labels which are the index of the owner *cell* of each *face* present in **faces**. The size of this list is equal to the size of faces list.
- **neighbor** which contains a list of labels which are the index to the neighbor cell of the faces present in **faces**. The size of this list is smaller than the size of **faces** as it only keeps the neighbors of the internal faces. The neighbor to the boundary faces is set to -1 by default. Therefore, the size of **neighbor** is equal to the number of internal faces.
- **boundary** which includes all the boundaries as their type, name, start face index in the **faces** files and number of faces.
- **zone** files may also be present if some blocks are specified as zones.

The important fact here is that the **faces** file is set up in such a way that, first all the internal faces are written and then the boundary *patchFields* are written as continuous blocks of faces in the **faces** file. This means that any boundary name corresponds to a single continuous slice of the faces list, it has one start face index and a size.
In order to split up a perforated plate into two patchFields we have to reorder the face list so that all the faces which together build up the hole part of the plate constitute a continuous slice of the face list, and the same rule applies for the faces building the wall part of the plate.

4.2 Getting started

The aim here is to modify the *blockMesh* utility so you should first copy *blockMesh* utility to your working directory.

```
cd $FOAM_USER_APP
cp -r $FOAM_APP/utilities/mesh/generation/blockMesh .
mv blockMesh blockMeshPP
cd blockMeshPP
```

4.3 Modifications

The main function is in **blockMeshApp.C** file where the **blockMeshDict** file (or any possible alternatives which is not the matter of concern here) is read as an input dictionary and then a *blockMesh* class object is constructed which is named **blocks**.

```
  Info<< nl << "Creating block mesh from\n    "
        << meshDictIoPtr->objectPath() << nl << endl;

    IOdictionary meshDict(meshDictIoPtr());
    blockMesh blocks(meshDict);
```

Then this object is not changed at all until a *polyMesh* class object is made from it which is named **mesh** as follows.

```
 polyMesh mesh
    (
        IOobject
        (
            regionName,
            runTime.constant(),
            runTime
        ),
        xferCopy(blocks.points()),           // could we re-use space?
        blocks.cells(),
        blocks.patches( ),
        patchNames,
        patchTypes,
        defaultFacesName,
        defaultFacesType,
        patchPhysicalTypes
);
```

This *polyMesh* constructor uses a number of **blocks** properties to make the mesh:

- *blocks.points()*
- *blocks.cells()*
- *blocks.patches()*

It also uses some *word*s and *wordList*s which are made earlier in the code from **blocks** object:

- **patchNames**
- **patchTypes**
- **defaultFacesName**
- **defaultFacesType**
- **patchPhysicalTypes**

Here is where these are defined in the code.

```
    wordList patchNames = blocks.patchNames();
    wordList patchTypes = blocks.patchTypes();
    word defaultFacesName = "defaultFaces";
    word defaultFacesType = emptyPolyPatch::typeName;
    wordList patchPhysicalTypes = blocks.patchPhysicalTypes();
```

Remember that this is the location of the code that you should step in for changes and will be mentioned later.
The rest of the code does not change the **mesh** and just sets up the zones if any specified, before writing the mesh to the files discussed earlier. All these suggest that we should manipulate those **blocks** object components that are used to construct the *polyMesh* in such a way that they match the topology of interest (this simply means that the perforated plate should be split in a topologically consistent way).
Let's start with passing the information about the perforated plates to the mesh generator through a dictionary. An obvious choice is to add it to the **blockMeshDict** dictionary. In this way you should first set up a **blockMeshDict** dictionary assuming the perforated plate boundary is a normal *patch type* and then add to the dictionary how to split it into a *wall* and a *patch.* The splitting instructions is given under the keyword **PPInfo** to the utility. Following this keyword comes a list of dictionaries, each describing one of the perforated plates in the geometry in terms of:

- The name of the boundary condition to be considered as a perforated plate. Remember we have already set this boundary as a *patch type*, this is the name of that patch and comes under the keyword **PPOldPatchName**.
- The name of the *polyPatch* you want to give to the *wall type* patch extracted from the perforated plate. This comes under the keyword **PPWallPatchName**.
- The name of the patch you want to give to the *patch type* patch extracted from the perforated plate representing the holes. This comes under the keyword **PPHolesPatchName**.
- A list of vectors representing the center of holes on the plate under the keyword **PPHoles-Centers**.
- A list of scalars representing the radii of the holes on the plate under the keyword **PP-HolesRadii**.

This is an example of a **blockMeshDict** where a perforated plate with three holes is made. The *wall type* patch representing the plate wall section is named **PP1Wall** and the holes part will be a *patch type* patch named **PP1Holes**. Notice that the plate was first defined as a *patch type* **PP1**.

```
.
.
.
patches
(
    patch PP1
    (
      (0 22 23 1)
    );
.
.
.
);
.
.
.
PPInfo
(
    {
        // Name of new patch
        PPOldPatchName        PP1;
        PPWallPatchName       PP1Wall;
        PPHolesPatchName      PP1Holes;
        PPHolesCenters  ((-0.0206 0.008 0)(-0.0206 0.016 0)(-0.0206 0.022 0));
        PPHolesRadii    (0.004 0.002 0.002);

    }
);
```

Now you should step in the code in the location mentioned earlier (it is right before defining the *preservePatchNames* which is followed by construction of the *polyMesh* **mesh object**), read the perforated plate information and put them in appropriate lists. Add the following to the code:

```
 // reading in the PP information from dictionary
    PtrList<dictionary> patchSources(meshDict.lookup("PPInfo"));
    // declaring some lists to keep the PPInfo for later use
    List<word> PPOldPatchNames(patchSources.size());
    List<word> PPWallPatchNames(patchSources.size());
    List<word> PPHolesPatchNames(patchSources.size());
    List< List<vector> > PPCenters(patchSources.size());
    List< List<scalar> > PPRadii(patchSources.size());

    // filling up the above lists with value read from dictionary

    forAll(patchSources,iPP)
    {
       word PPOldPatchNametmp(patchSources[iPP].lookup("PPOldPatchName"));
       PPOldPatchNames[iPP]=PPOldPatchNametmp;
       word PPWallPatchNametmp(patchSources[iPP].lookup("PPWallPatchName"));
       PPWallPatchNames[iPP]=PPWallPatchNametmp;
       word PPHolesPatchNametmp(patchSources[iPP].lookup("PPHolesPatchName"));
       PPHolesPatchNames[iPP]=PPHolesPatchNametmp;
       List<vector> PPCenterstmp(patchSources[iPP].lookup("PPCenters"));
       PPCenters[iPP]=PPCenterstmp;
       List<scalar> PPRadiitmp(patchSources[iPP].lookup("PPRadii"));
       PPRadii[iPP]=PPRadiitmp;
     }
```

Every patch representing a perforated plate is split into two patches so the number of patches of the final mesh would be the number of original patches plus the number of perforated plates. The number of original patches is found from *blocks.patches().size()* and the number of perforated plates is the size of the list followed by the **PPInfo** keyword, *patchSources.size()*. Use these to set up new objects which will be filled with information needed to construct the final mesh. The code follows:

```
// declaring some lists which will contain the modified information about the blockMesh object
    List< List<face> > oldPatches(blocks.patches());
    List< List<face> > newPatches(oldPatches.size()+patchSources.size());
    wordList newPatchNames(oldPatches.size()+patchSources.size());
    wordList newPatchTypes(oldPatches.size()+patchSources.size());
    wordList newPatchPhysicalTypes(oldPatches.size()+patchSources.size());
```

There are two functions needed to be defined before you can continue. You need the center of each face on the perforated plate patch to be able to determine if it is located on a hole or on the wall of a perforated plate. This information is not available from *blockMesh.patches()* so you need a function to do it. This function can be copied from **$FOAM\/src/OpenFOAM/meshes/primitiveMesh/primitiveMeshFaceCe**
and then modified accordingly. Add this to the end of the code as the function definition:

```
vector faceCenterPP
(
 const pointField& p,
 const face& facePP
 )
{
  vector fCenter = vector::zero;
  label nPoints = facePP.size();

  // If the face is a triangle, do a direct calculation for efficiency
  // and to avoid round-off error-related problems
  if (nPoints == 3)
    {
      fCenter  = (1.0/3.0)*(p[facePP[0]] + p[facePP[1]] + p[facePP[2]]);
    }
  else
    {
      vector sumN = vector::zero;
      scalar sumA = 0.0;
      vector sumAc = vector::zero;

      point fCent = p[facePP[0]];
      for (label pi = 1; pi < nPoints; pi++)
{
  fCent += p[facePP[pi]];
}

      fCent /= nPoints;

      for (label pi = 0; pi < nPoints; pi++)
{
  const point& nextPoint = p[facePP[(pi + 1) % nPoints]];

  vector c = p[facePP[pi]] + nextPoint + fCent;
  vector n = (nextPoint - p[facePP[pi]])^(fCent - p[facePP[pi]]);
  scalar a = mag(n);

  sumN += n;
  sumA += a;
  sumAc += a*c;
}

      fCenter = (1.0/3.0)*sumAc/(sumA + VSMALL);
    }
  return fCenter;
}
```

You also need to declare it before you can use it so add the following to the beginning of the file:

```
    // function to calculate a face center
vector faceCenterPP
(
 const pointField& p,
 const face& facePP
 );
```

The next function is one that can recognize if a face is in a hole or on the wall of a perforated plate. Add this to the beginning as the declaration

```
     // function to evaluate if a given face is in a hole or the wall part of a perforatedPlate
bool isHole
(
 const vector& faceCenterPP,
 const List<vector>& PPcenters ,
 const List<scalar>& PPradii
 );
```

And add the function definition at the end:

```
  bool isHole
(
 const vector& faceCenterPP,
 const List<vector>& PPcenters ,
 const List<scalar>& PPradii
)
{
  bool ishole=false;
  forAll(PPradii,iHole)
      {
if(mag(faceCenterPP-PPcenters[iHole])<=PPradii[iHole])
      {
ishole=true;
      }
}

  return ishole;
}
```

Now go back to the point you were in the code and add the following. To understand what this does just follow the comments in the code.

```
 if (patchSources.size()) //if any perforated plate exists
{
  label PPWallFaceIndex=0;
  label PPHolesFaceIndex=0;
  label totPatchIndex=0;

        forAll(oldPatches, patchI)// go through all original patches
                                 //(patches defined earlier considering the palte as a single patch)
  {
            bool isPP = false;
    label PPno(-1);

            forAll(PPOldPatchNames,ii)// see if the original patch name exists in the list of perforat
      {
if(patchNames[patchI]==PPOldPatchNames[ii])
  {
    isPP = true;
    PPno = ii;
  }
      }

    if(!isPP)// if the current original patch is not a perforated plate just copy it as a member of mo
      {
newPatches[totPatchIndex] = oldPatches[patchI];
newPatchNames[totPatchIndex] = patchNames[patchI];
newPatchTypes[totPatchIndex] = patchTypes[patchI];
newPatchPhysicalTypes[totPatchIndex] = patchPhysicalTypes[patchI];

totPatchIndex++;//increase the number of patches by one
      }
            else//if the current original path is a perforated plate
      {
// define two new patches one a wall type and the other a patch type
newPatchNames[totPatchIndex] = PPWallPatchNames[PPno];
newPatchTypes[totPatchIndex] = "wall";
newPatchPhysicalTypes[totPatchIndex] = patchPhysicalTypes[patchI];
                totPatchIndex++;
                newPatchNames[totPatchIndex] = PPHolesPatchNames[PPno];
newPatchTypes[totPatchIndex] = "patch";
newPatchPhysicalTypes[totPatchIndex] = patchPhysicalTypes[patchI];
                totPatchIndex++;

label PPWallFaceCount=0;
                label PPHolesFaceCount=0;
                forAll(oldPatches[patchI],ii)// go through all of the faces in the current original pa
                                             // which is recognized as a perforated plate
  {
    // count the number of faces in these two new patches by checking the center of every face
                    // to find if it is in a hole on the perforated plate or not use isHole function
    if(isHole(faceCenterPP(blocks.points(),oldPatches[patchI][ii]),PPCenters[PPno],PPRadii[PPno]))
      {
PPHolesFaceCount++;
      }
    else
      {
PPWallFaceCount++;
      }
  }
                //set up two lists resembling the new patches that should be made from PP
                //the size of these two lists are know from counting the faces on holes and walls
                List<face> PPWall(PPWallFaceCount);
List<face> PPHoles(PPHolesFaceCount);
label PPWallFaceIndex=0;
                label PPHolesFaceIndex=0;
// fill up these face lists from the original patch
                forAll(oldPatches[patchI],ii)
  {
    if(isHole(faceCenterPP(blocks.points(),oldPatches[patchI][ii]),PPCenters[PPno],PPRadii[PPno]))
      {
PPHoles[PPHolesFaceIndex]=oldPatches[patchI][ii];
PPHolesFaceIndex++;
      }
    else
      {
PPWall[PPWallFaceIndex]=oldPatches[patchI][ii];
PPWallFaceIndex++;
      }

  }
newPatches[totPatchIndex-2] = PPWall;//add the wall face list of the perforated plate
                                                   //as a new patch to the new patch list we are set
newPatches[totPatchIndex-1] = PPHoles;//add the patch face list to the new patch list
                }
  }
      }
```

At this point you have enough information to construct the mesh. Add the following to the code and follow the comments.

```
 preservePatchTypes
      (
       runTime,
       runTime.constant(),
       polyMeshDir,
       patchNames,
       patchTypes,
       defaultFacesName,
       defaultFacesType,
       patchPhysicalTypes
       );

    // construct the polymesh type object mesh from this modified blockMesh object instead of the orig
    // the original construction is commented below
 polyMesh mesh
    (
        IOobject
        (
            regionName,
            runTime.constant(),
            runTime
        ),
        xferCopy(blocks.points()),           // could we re-use space?
        blocks.cells(),
        newPatches,
        newPatchNames,
        newPatchTypes,
        defaultFacesName,
        defaultFacesType,
        newPatchPhysicalTypes
);

    /*polyMesh mesh
    (
        IOobject
        (
            regionName,
            runTime.constant(),
            runTime
        ),
        xferCopy(blocks.points()),           // could we re-use space?
        blocks.cells(),
        blocks.patches(),
        patchNames,
        patchTypes,
        defaultFacesName,
        defaultFacesType,
        patchPhysicalTypes
);*/
```

4.4 compiling

Before you compile this new application modify the Make/files file and change the name of the executable and then simply compile.

```
EXE = $(FOAM_USER_APPBIN)/blockMeshPP
```

5 Examples

5.1 as boundary

You can use the pitzDaily tutorial as a validation case but it is more clear if you change it to a three dimensional case. Start by copying the tutorial to your run directory:

```
run
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
mv pitzDaily pitzDaily3d_asBoundary
cd  pitzDaily3d_asBoundary
```

Now open the **blockMeshDict** file and make the mesh 3d. Just change the z bounds from (-0.5,0.5) to (-20 ,20) in vertices and set the number of cells in z direction to 20 in all the blocks. You should also merge the **frontAndBack** with **lowerwall** patch into a single patch. The inflow boundary is our target to be changed to a perforated plate so it can be defined either as a *wall* or a *patch* so you can keep it as it is.
Then open the **0/U** file and change the inlet patch according to:

```
    inlet
    {
        type            perforatedPlateFixedValue;
        value           uniform (10 0 0);
        PPHolesValue    uniform (10 0 0);
        PPWallValue     uniform (0 0 0);
        PPHolesCenters  ((-0.0206 0.017 -0.015) (-0.0206 0.017 0) (-0.0206 0.017 0.015) );
        PPHolesRadii    (0.003 0.007 0.003 );
    }
```

This will set a *uniform* velocity of 10 in x direction on the holes and zero on the wall of the perforated plate. The plate will have three holes as specified above. Note that all the other files in the **0/** directory should be changed as the way they are now will give incorrect results. Some are even physically wrong, for example the **k** has a value on the wall now. However for the matter of validating the boundary condition these can be neglected and the above change is enough. Do not forget to delete the **frontAndBack** patch from all the files in **0/** directory. You can now run the case and have a look at the U velocity at the inlet boundary, figure 1.2.

```
blockMesh
simpleFoamPP
```

5.2 Modified blockMesh

Start by copying the tutorial to your run directory:

```
run
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
mv pitzDaily pitzDaily3d_blockMeshPP
cd  pitzDaily3d_blockMeshPP
```

Then open the **constant/polyMesh/blockMeshDict**, change to 3d as you did before and merge the **frontAndBack** with **lowerWall**. Then change the **inlet** patch to

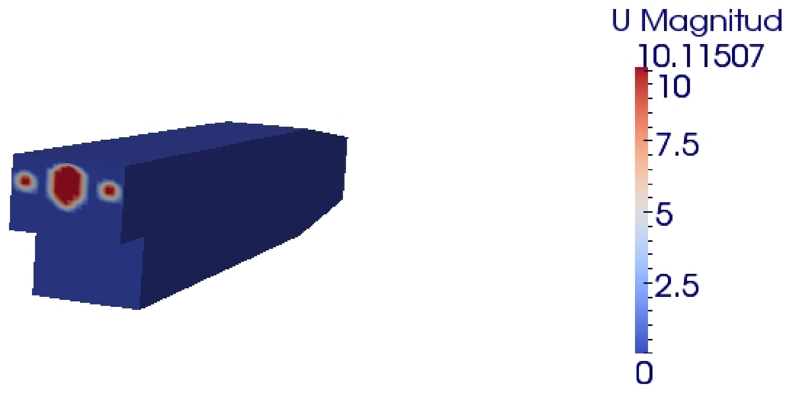

Figure 2: Velocity magnitude at the inlet showing the perforated plate alternative 1

```
patch PP1
    (
        (0 22 23 1)
        (1 23 24 2)
        (2 24 25 3)
    )
```

and add the following to the end of the file:

```
PPInfo
(
    {
        PPOldPatchName          PP1;
        PPWallPatchName         PP1Wall;
        PPHolesPatchName        PP1Holes;
        PPCenters   ((-0.0206 0.017 -0.015) (-0.0206 0.017 0) (-0.0206 0.017 0.015) );
        PPRadii     (0.003 0.007 0.003 );
    }
);
```

Now run *blockMeshPP* application and have a look at **/constant/polymesh/boundary**, the following patches are present now:

```
5
(
    PP1Wall
    {
        type            wall;
        nFaces          522;
        startFace       715675;
    }
    PP1Holes
    {
        type            patch;
        nFaces          78;
        startFace       716197;
    }
    outlet
    {
        type            patch;
        nFaces          1140;
        startFace       716275;
    }
    upperWall
    {
        type            wall;
        nFaces          4460;
        startFace       717415;
    }
    lowerWall
    {
        type            wall;
        nFaces          29450;
        startFace       721875;
    }
)
```

Now go to the **0/** directory and specify your boundary conditions. Remember to delete **frontAndBack** in all the files. An example for **0/U** would be:

```
boundaryField
{
    PP1Holes
    {
        type            fixedValue;
        value           uniform (10 0 0);

    }
    PP1Wall
    {
        type            fixedValue;
        value           uniform (0 0 0);

    }


    outlet
    {
        type            zeroGradient;
    }

    upperWall
    {
```

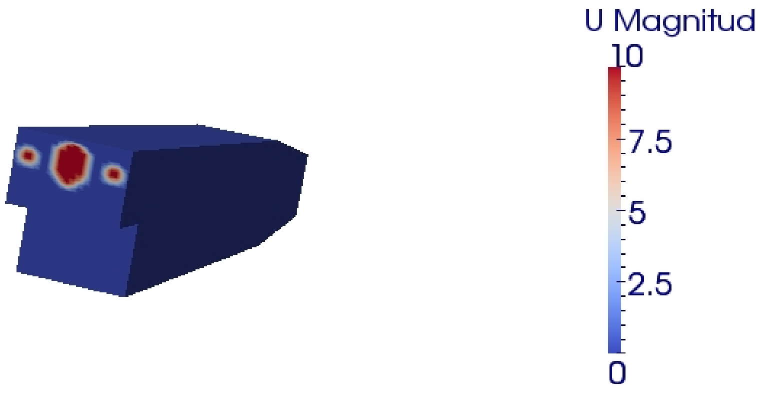

Figure 3: Velocity magnitude at the inlet showing the perforated plate alternative 2

```
        type            fixedValue;
        value           uniform (0 0 0);
    }

    lowerWall
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

}
```

Then run the original *simpleFoam* application. Figure 1.3 shows the results which are identical to the previous implementation.

```
blockMeshPP
simpleFoam
```