Cite as: Josefsson, E.: Methods for wheel rotation modelling. In Proceedings of CFD with OpenSource Software, 2020, Edited by Nilsson. H., `http://dx.doi.org/10.17196/OS_CFD#YEAR_2020`

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Methods for wheel rotation modelling

Developed for OpenFOAM-v2006

Author:
Erik Josefsson
Chalmers University of Technology
erik.josefsson@chalmers.se

Peer reviewed by:
Eleanor Harvey
Simone Sebben
Saeed Salehi

January 15, 2021

Learning outcomes

This section aims at describing what this tutorial will cover. The learning outcomes are split into four categories which are presented below.

The reader will learn:

How to use it:

- How to use the rotating wall boundary condition.
- How to use MRF to simulate the flow inside the lateral grooves of a tyre.

The theory of it:

- About different methods used for wheel rotation modelling.
- About the benefits and drawbacks of different wheel rotation modelling methods, and how the methods can be combined into hybrid models.

How it is implemented:

- How the rotating wall boundary condition is implemented.
- How the MRF method is implemented.
- How the rotating mesh is implemented.

How to modify it:

- How to modify the MRF method to read additional input in the dictionary and use the new input to modify the affect from MRF on the flow.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- How to run standard document tutorials, like the motorbike tutorial, in OpenFOAM.
- Have a fundamental understanding of CFD.

Contents

Nomenclature

Acronyms

CFD	Computational Fluid Dynamics
MRF	Multiple Reference Frame
MRFg	Multiple Reference Frame grooves
RW	Rotating Wall

English symbols

$\lvert\vec{n}\rvert$	Cell face area	m^2
$\vec{n}$	Cell face normal vector	m
$\vec{r}$	Cartesian spatial vector	m
$\vec{u}$	Cartesian velocity vector	m/s
$\vec{u}_f$	Interpolated face velocity vector	m/s
p	Pressure	Pa

Greek symbols

ν	Kinematic viscosity	m^2/s
ω	Angular velocity magnitude	1/s
ϕ	Face flux	m^3/s
ρ	Density	kg/m^3
$\vec{\Omega}$	Angular velocity vector	1/s

Subscripts

I	inertial
R	relative
abs	absolute

Chapter 1

Introduction

1.1 Background

The transport sector is one of the main contributors to the global CO_2 emissions. Therefore, there is a need for developing new, more environmentally friendly modes of transport. Within the transport sector a significant part of the emissions comes from road vehicles in general, and cars in particular. Therefore, car companies are continuously striving to reduce the energy consumption of their vehicles, both to reduce the emissions from cars with internal combustion engines, as well as allowing for longer range in new, battery electric vehicles.

For a typical passenger car at velocities above approximately 60 km/h the aerodynamic drag is the largest resisting force to overcome. Furthermore, around 25% of the aerodynamic drag can be attributed to the wheels. CFD is today used extensively in the development of new vehicles since it, among other things, reduces lead time and can be used already in the early stages of a project. Therefore it is important to be able to accurately replicate the geometry and boundary conditions of the wheels in CFD. Improved CFD simulations will allow for a greater understanding of the flow around wheels, which in turn will allow for new, energy efficient vehicles.

1.2 Wheel rotation modelling

When modelling wheels in CFD there are two main challenges, replicating the geometry and applying representative boundary conditions. The focus of this work will be the boundary conditions. However, the boundary conditions are influenced by how the geometry is replicated, hence some background will be given on the geometry aspect as well.

1.2.1 Geometry replication

Figure 1.1 shows the two main parts of a wheel, the tyre and the rim, along with some of the nomenclature that will be used in this tutorial. Since the tyre deforms under the load of the vehicle, creating a contact patch (or footprint) to the ground, the challenges of replicating the geometry is mainly in replicating the tyre. The most straightforward method, which is also widely used, is to take the CAD of an undeformed (unloaded) tyre and place it such that the lower part of the tyre is cut by the ground, creating a contact patch. However, this representation of the tyre is not physical, creating an unrealistic contact patch as well as neglecting the bulge at the side of the tyre above the contact patch. It has been shown that the flow field predicted in CFD can be improved significantly by deforming the geometry, replicating the actual contact patch and bulge.

1.2.2 Boundary conditions

In order to replicate the rotation of the wheels special care needs to be taken when selecting the boundary conditions. The most straightforward approach is to use a rotating wall boundary condi-

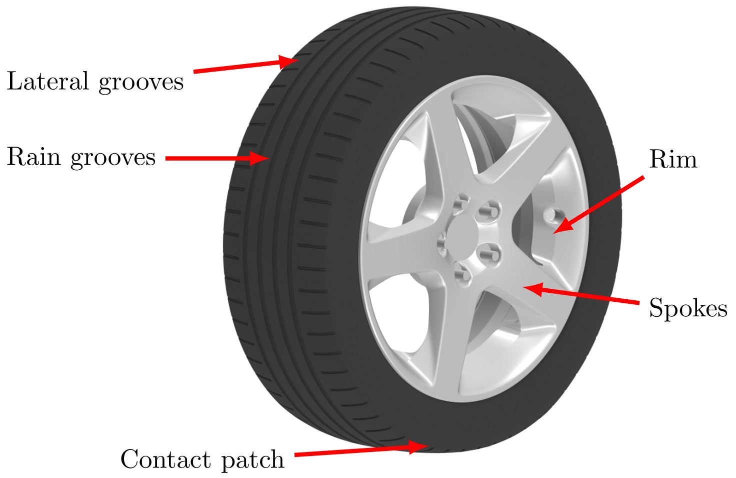


Figure 1.1: Names used for the different parts of the wheel.

tion, which applies a velocity to the wheel according to a rotation specified by the user.

1.2.2.1 Rotating wall

For a simplified case, with a slick tyre and a closed rim (no large opening between the spokes allowing flow through the rim) this boundary condition has been shown to generate representative results. However, a significant drawback is that the rotational velocity can only be applied tangentially to the surface. This means that regions where the surfaces are largely normal to the desired velocity vector will not be represented accurately, since a large part of the velocity will be lost when projecting the vector to the surface. For a wheel such regions exists between the spokes of the rim as well as in some of the grooves in the tyre surface,

1.2.2.2 Multiple reference frame (MRF)

In the MRF method cells are placed in a rotating reference frame. This allows for velocity components normal to the surface, resolving the problem with rotating wall. The incompressible Navier-Stokes equations are modified to

$$\begin{gathered}\frac{\partial \vec{u}_R}{\partial t} + \frac{\partial \vec{\Omega}}{\partial t} \times \vec{r} + \nabla \cdot (\vec{u}_R \otimes \vec{u}_I) + \vec{\Omega} \times \vec{u}_I = -\nabla (p/\rho) + \nu \nabla \cdot \nabla (\vec{u}_I) \\ \nabla \cdot \vec{u}_I = 0,\end{gathered} \tag{1.1}$$

where I and R denotes the inertial and relative reference frame, respectively, $\vec{\Omega}$ is the rotational vector and $\vec{r}$ is the vector from the origin of rotation to the cell [1]. Since the geometry is not actually moved using the MRF method the results are dependent on the wheel position, shown in Figure 1.2. This was discussed by Landström et al. [2] and has been shown to be a significant drawback of the MRF model.

Additionally, it has been shown that the choice of MRF region can severely effect the flow. Assuming a steady flow the time derivatives in Eq. (1.1) vanishes. If also assuming uniform flow, $\nabla \vec{u}_I = 0$, Eq. (1.1) simplifies to

$$\vec{\Omega} \times \vec{u}_I = -\nabla (p/\rho) . \tag{1.2}$$

Hence, when $\vec{\Omega} \times \vec{u}_I \neq 0$, an undesirable pressure gradient will be introduced. In the case of a wheel the flow is largely orthogonal to the axis of rotation, meaning that this pressure gradient is likely to occur. This effect was illustrated and discussed in greater detail by Hobeika [3].

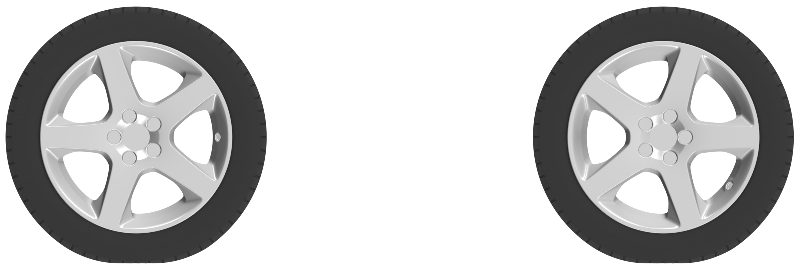

(a) Original position (b) Rim rotated 36°

Figure 1.2: Different positions for the rim, resulting in different solutions when using MRF.

1.2.2.3 Rotating mesh

In the rotating mesh method parts of the mesh are physically rotated at every time step. Thus it is the only method among the ones presented here that achieves an actual mesh rotation, hence it can be seen as the most realistic way to model the rotation of wheels. However there are some drawbacks of the method. Naturally an unsteady simulation is needed when using a rotating mesh. Furthermore the interface between the rotating and the stationary region needs to be updated at every time step, increasing the computational time. Furthermore, one of the largest limitations of the rotating mesh method is that it is not suitable for modelling tyres that are in contact with the ground, where they lose their circular shape. As described above a contact patch, as well as a bulge above the contact patch, is created when the tyre sits on the ground. This deformation removes the possibility to use rotating mesh for the tyre since the tyre is not rationally symmetric. Instead rotating mesh is used only for the rim, where it is considered to be the most suitable method.

1.2.2.4 Hybrid methods

As an attempt to leverage the benefits of the different wheel rotation methods hybrid methods have been proposed. This work will focus on describing the method proposed by Hobeika and Sebben [4]. There, a rotating mesh is used for the rim and a rotating wall condition is used on the tyre surface. However, to account for the grooves in the tyre surface with faces normal to the desired velocity, which rotating wall can not handle, MRF zones, denoted MRFg, are introduced in the grooves. This solves the problems of the rotating wall boundary condition and allows for a surface normal velocity component. Since the tyre grooves are a highly repetitive pattern the drawbacks of MRF not moving the geometry and only representing one physical position are considered to be small.

Chapter 2

Rotation methods in OpenFOAM

In this section different techniques for simulating rotation implemented in OpenFOAM will be presented. For more details, especially about moving meshes, see the training by Nilsson [5].

2.1 Rotating wall

The rotating wall boundary condition can be found in

`$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/rotatingWallVelocity`

The boundary condition can be used by adding the following to the `0/U` file:

Example of rotating wall in boundary condition file `U`

```
tyre
{
    type            rotatingWallVelocity;
    origin          (0 0 0);
    axis            (0 1 0);
    omega           -194.45;
}
```

Here the `rotatingWallVelocity` boundary condition is applied to the patch called `tyre`. The origin of the rotation is specified along with the axis of rotation, `axis`, and the angular velocity, `omega`. Investigating `rotatingWallVelocityFvPatchVectorField.H` it can be seen that this corresponds to constructing an object using the constructor taking a patch, internal field and dictionary as inputs, which is the method this description will focus on. The construction of the object is implemented in `rotatingWallVelocityFvPatchVectorField.C` as:

The constructor in `rotatingWallVelocityFvPatchVectorField.C`

```
Foam::rotatingWallVelocityFvPatchVectorField::
rotatingWallVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchField<vector>(p, iF, dict, false),
    origin_(dict.lookup("origin")),
    axis_(dict.lookup("axis")),
    omega_(Function1<scalar>::New("omega", dict))
{
    if (dict.found("value"))
    {
        fvPatchField<vector>::operator=
        (
            vectorField("value", dict, p.size())
        );
    }
    else
    {
        // Evaluate the wall velocity
        updateCoeffs();
    }
}
```

It can be seen that a `fixedValueFvPatchVectorField` is created. Furthermore the values of `origin_`, `axis_` and `omega_` are read from the dictionary. If a dictionary field `value` has not been specified, as is the case here, the function `updateCoeffs` is evaluated. The function contains the following:

`updateCoeffs` in `rotatingWallVelocityFvPatchVectorField.C`

```
void Foam::rotatingWallVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const scalar t = this->db().time().timeOutputValue();
    scalar om = omega_->value(t);

    // Calculate the rotating wall velocity from the specification of the motion
    const vectorField Up
    (
        (-om)*((patch().Cf() - origin_) ^ (axis_/mag(axis_)))
    );

    // Remove the component of Up normal to the wall
    // just in case it is not exactly circular
    const vectorField n(patch().nf());
    vectorField::operator=(Up - n*(n & Up));

    fixedValueFvPatchVectorField::updateCoeffs();
}
```

If the boundary condition is already updated the function call returns without doing anything. If not, the value of `omega_`, which can be time dependant, is updated. The rotating wall velocity `Up` is then calculated as

$$\vec{u}_p = -\omega\left(\vec{r}_f - \vec{r}_0\right) \times \frac{\vec{\Omega}}{|\vec{\Omega}|}, \tag{2.1}$$

where ω is the angular velocity, $\vec{r}_f$ the centre of the face, $\vec{r}_0$ the origin of the rotation and $\vec{\Omega}$ the rotational axis. One important note here is that the vector for the rotational axis is normalised in the calculation, hence the axis can not be used to control the magnitude of the rotational velocity.

After calculating the rotational wall velocity the normals of the patch faces are found. These are then used to remove the normal component of the velocity, which is done by projecting the velocity to the normal direction of the face and removing the resulting vector from `Up` as

$$\vec{u}_{p,\text{new}} = \vec{u}_p - \vec{n}\left(\vec{n} \cdot \vec{u}_p\right), \tag{2.2}$$

where $\vec{n}$ is the normal of the face. This last step confirms that rotating wall therefore is not suitable for cases where the desired velocity component is largely normal to the surface. However, here it should be noted that this is the reasonable approach. If the boundary condition would keep the component normal to the surface that would represent a flow through the surface, which would not be valid.

2.2 Multiple reference frame (MRF)

The MRF method is implemented as a part of `finiteVolume` and can be found in:

`$FOAM_SRC/finiteVolume/cfdTools/general/MRF`

Here the description of the implementation will be split into two parts. Firstly it will be described how the initial setup of the MRF zones is done. This will be followed by a description of how the MRF affects the results when solving the equations.

2.2.1 Initial setup

The MRF method is implemented in a number of different solvers. Here the implementation will be described using the solver `pimpleFoam` as a starting point. However, the general principles should be the same for any solver. The solver can be found in:

`$FOAM_APP/solvers/incompressible/pimpleFoam`

Below the first part of `pimpleFoam.C` is shown (excluding the header).

The beginning of `pimpleFoam.C`

```
 77 #include "fvCFD.H"
 78 #include "dynamicFvMesh.H"
 79 #include "singlePhaseTransportModel.H"
 80 #include "turbulentTransportModel.H"
 81 #include "pimpleControl.H"
 82 #include "CorrectPhi.H"
 83 #include "fvOptions.H"
 84 
 85 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
 86 
 87 int main(int argc, char *argv[])
 88 {
 89     argList::addNote
 90     (
 91         "Transient solver for incompressible, turbulent flow"
 92         " of Newtonian fluids on a moving mesh."
 93     );
 94 
 95     #include "postProcess.H"
 96 
 97     #include "addCheckCaseOptions.H"
 98     #include "setRootCaseLists.H"
 99     #include "createTime.H"
100     #include "createDynamicFvMesh.H"
101     #include "initContinuityErrs.H"
102     #include "createDyMControls.H"
103     #include "createFields.H"
104     #include "createUfIfPresent.H"
105     #include "CourantNo.H"
106     #include "setInitialDeltaT.H"
107 
108     turbulence->validate();
```

In this code there are no signs of the implementation of MRF. Instead MRF can be found in `createFields.H`, which is located in the same directory, and is called by `pimpleFoam.C` on line 103. At the end of `createFields.H` the line `#include "createMRF.H"` is found. Searching in the OpenFOAM directory this file can be found in

`$FOAM_SRC/finiteVolume/cfdTools/general/include/createMRF.H`

This file contains one single line `IOMRFZoneList MRF(mesh);`, creating an object `MRF` of the class `IOMRFZoneList`. The constructor for this object is found in

`$FOAM_SRC/finiteVolume/cfdTools/general/MRF/IOMRFZoneList.C`

and is shown below.

The constructor in `IOMRFZoneList.C`

```
Foam::IOMRFZoneList::IOMRFZoneList
(
    const fvMesh& mesh
)
:
    IOdictionary(createIOobject(mesh)),
    MRFZoneList(mesh, *this)
{}
```

It can be seen that an `IOdictionary` is created using the function `createIOobject` and afterwards a `MRFZoneList` is created. Starting with the `IOdictionary` the code is found above in the same file and contains:

The function `createIOobject` in `IOMRFZoneList.C`

```
Foam::IOobject Foam::IOMRFZoneList::createIOobject
(
    const fvMesh& mesh
) const
{
    IOobject io
    (
        "MRFProperties",
        mesh.time().constant(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    );

    if (io.typeHeaderOk<IOdictionary>(true))
    {
        Info<< "Creating MRF zone list from " << io.name() << endl;

        io.readOpt() = IOobject::MUST_READ_IF_MODIFIED;
        return io;
    }
    else
    {
        Info<< "No MRF models present" << nl << endl;

        io.readOpt() = IOobject::NO_READ;
        return io;
    }
}
```

An `IOobject` is created reading from the file `MRFProperties`. The function then looks for the file. If it is found this is confirmed in the output and the setting for when to read the file is changed to only read if there has been an update. The `IOobject` is then returned. If the file can not be found this is reported to the user via the output, the `IOobject` is set to not read the file and the `IOobject` is returned. Below an example of the file `MRFProperties` is given.

`MRFProperties`

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
```

```
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      MRFProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

MRF1
{
    cellZone    mrfgZone;
    active      yes;

    // Fixed patches (by default they 'move' with the MRF zone)
    nonRotatingPatches ();

    origin          (0 0 0);
    axis            (0 1 0);
    omega           -194.45;
}

// ************************************************************************* //
```

Similarly to the rotating wall boundary condition an origin and an axis of rotation is defined along with an angular velocity for every `MRFZone`. Additionally the `cellZone` for which the MRF should be applied is specified. Finally there is also the possibility to omit patches from the MRF using `nonRotatingPatches` and deactivating the MRF by using `active`.

Returning to the constructor of `IOMRFZoneList.C` a `MRFZoneList` is created. This is done in `MRFZoneList.C`, which is found in the same directory. The constructor for this object is shown below.

The constructor in `MRFZoneList.C`

```
Foam::MRFZoneList::MRFZoneList
(
    const fvMesh& mesh,
    const dictionary& dict
)
:
    PtrList<MRFZone>(),
    mesh_(mesh)
{
    reset(dict);

    active(true);
}
```

Here it can be noted that the constructor takes a `fvMesh` and a `dictionary` as input. However, in `IOMRFZoneList.C` an `fvMesh` and the object itself was used to call the function. This is possible since the `IOMRFZoneList` inherits `IOdictionary`, as can be seen in `IOMRFZoneList.H`. A list of pointers to the MRF zones is constructed along with a reference to the mesh. The constructor then calls two functions `reset` and `active`. The latter of these checks if any MRF zone is activated. If not, this is reported in the output. The function `reset` creates the MRF zones specified in the dictionary according to the code below.

The function `reset` in `MRFZoneList.C`

```
void Foam::MRFZoneList::reset(const dictionary& dict)
{
    label count = 0;
    for (const entry& dEntry : dict)
    {
        if (dEntry.isDict())
        {
```

```
 81              ++count;
 82          }
 83      }
 84
 85      this->resize(count);
 86
 87      count = 0;
 88      for (const entry& dEntry : dict)
 89      {
 90          if (dEntry.isDict())
 91          {
 92              const word& name = dEntry.keyword();
 93              const dictionary& modelDict = dEntry.dict();
 94
 95              Info<< "    creating MRF zone: " << name << endl;
 96
 97              this->set
 98              (
 99                  count++,
100                  new MRFZone(name, mesh_, modelDict)
101              );
102          }
103      }
104  }
```

Firstly, the number of entries in the dictionary is checked using the first loop (rows 77-83). Knowing the number of entries the list of pointers is resized accordingly. This is followed by the function looping through all dictionary entries, taking the name and the settings of each zone specified in the dictionary and constructing a new `MRFZone`, which is then added to the list of pointers (rows 88-103). Until the loop reading the dictionary entries and creating `MRFZone` the parts of the code described will be executed by `pimpleFoam` regardless of MRF is used in the simulation or not. However, for a case without a `MRFProperties` file (or a file without any zones specified) the number of entries in the dictionary would be zero. Hence the code inside the second loop would not the executed.

Investigating `MRFZone.C`, which also is in the same directory, the constructor contains the following:

The constructor of `MRFZone.C`

```
Foam::MRFZone::MRFZone
(
    const word& name,
    const fvMesh& mesh,
    const dictionary& dict,
    const word& cellZoneName
)
:
    mesh_(mesh),
    name_(name),
    coeffs_(dict),
    active_(coeffs_.getOrDefault("active", true)),
    cellZoneName_(cellZoneName),
    cellZoneID_(),
    excludedPatchNames_
    (
        coeffs_.getOrDefault<wordRes>("nonRotatingPatches", wordRes())
    ),
    origin_(coeffs_.get<vector>("origin")),
    axis_(coeffs_.get<vector>("axis").normalise()),
    omega_(Function1<scalar>::New("omega", coeffs_))
{
    if (cellZoneName_ == word::null)
    {
        coeffs_.readEntry("cellZone", cellZoneName_);
    }

    if (!active_)
267     {
268         cellZoneID_ = -1;
269     }
270     else
271     {
272         cellZoneID_ = mesh_.cellZones().findZoneID(cellZoneName_);
273 
274         const labelHashSet excludedPatchSet
275         (
276             mesh_.boundaryMesh().patchSet(excludedPatchNames_)
277         );
278 
279         excludedPatchLabels_.setSize(excludedPatchSet.size());
280 
281         label i = 0;
282         for (const label patchi : excludedPatchSet)
283         {
284             excludedPatchLabels_[i++] = patchi;
285         }
286 
287         bool cellZoneFound = (cellZoneID_ != -1);
288 
289         reduce(cellZoneFound, orOp<bool>());
290 
291         if (!cellZoneFound)
292         {
293             FatalErrorInFunction
294                 << "cannot find MRF cellZone " << cellZoneName_
295                 << exit(FatalError);
296         }
297 
298         setMRFFaces();
299     }
300 }
```

In the code it can be seen that, after creating a reference to the mesh and setting the name, a dictionary (can be seen in `MRFZone.H`) named `coeffs_` is constructed from the dictionary in the call to the constructor. `coeffs_` is then used to set `active_`, `excludedPatchNames_`, `origin_`, `axis_` and `omega_`. If included in the call to the constructor, `cellZoneName_` is set. Otherwise this is later read from `coeffs_` (rows 261-264). Moving on the `cellZoneId_` is set. If the zone is not activated the value is set to -1 (rows 266-269). Otherwise it is found using the `cellZoneName_` and the function `findZoneId` (row 272). Here it can be noted that if the name can not be found the function returns -1, similarly to if the zone is not activated. Next a list with the labels of the excluded patches is constructed from `excludedPatchNames_` (rows 274-285). Afterwards a check is performed to see that the cell zone was found, if not the solver will exit. If the zone was found the function `setMRFFaces` is called.

The function `setMRFFaces` arranges the faces in each MRF zone in three categories. These are `internalFaces_`, `includedFaces_` and `excludedFaces_`. The different type of faces will later be used to set the fluxes. The process of arranging the faces will not be described in more detail here, the code can be found in Appendix A.1. It should however be noted that if `debug` is activated for `MRFZone` in the global `controlDict` the function will create `faceSets` for the three different categories.

2.2.2 Implementation during solving

Now all MRF zones have been constructed and the execution returns to `pimpleFoam`. The next time MRF influences the solver is at the beginning of the `run` loop. There, if the mesh is changing `MRF.update()` will be executed before solving the equations. This update goes through the functions described in Section 2.2.1 and executes `setMRFFaces` for all cell zones. Next the velocity equation is solved, which is done in `UEqn.H`, as shown below.

First part of `UEqn.H` implemented in `pimpleFoam.C`

```
MRF.correctBoundaryVelocity(U);

tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(U) + fvm::div(phi, U)
  + MRF.DDt(U)
  + turbulence->divDevReff(U)
 ==
    fvOptions(U)
);
fvVectorMatrix& UEqn = tUEqn.ref();
```

Here MRF affects two parts of the code. Firstly, before solving the equations the boundary velocities are corrected for the MRF. Secondly, a term from the MRF is included in the left hand side of the velocity equation. Initially, `MRF.correctBoundaryVelocity(U)` will be studied. This function can, despite `MRF` being constructed as an object of the type `IOMRFZoneList`, be found in `MRFZoneList.C`, which is possible since `IOMRFZoneList` inherits `MRFZoneList`. Below the function `correctBoundaryVelocity` in `MRFZoneList` is shown.

The function `correctBoundaryVelocity` in `MRFZoneList.C`

```
void Foam::MRFZoneList::correctBoundaryVelocity(volVectorField& U) const
{
    forAll(*this, i)
    {
        operator[](i).correctBoundaryVelocity(U);
    }
}
```

It can be seen that this function in turns calls the function `correctBoundaryVelocity`, which now is located in `MRFZone.C`, for every `MRFZone`. This function is shown below.

The function `correctBoundaryVelocity` in `MRFZone.C`

```
void Foam::MRFZone::correctBoundaryVelocity(volVectorField& U) const
{
    if (!active_)
    {
        return;
    }

    const vector Omega = this->Omega();

    // Included patches
    volVectorField::Boundary& Ubf = U.boundaryFieldRef();

    forAll(includedFaces_, patchi)
    {
        const vectorField& patchC = mesh_.Cf().boundaryField()[patchi];

        vectorField pfld(Ubf[patchi]);

        forAll(includedFaces_[patchi], i)
        {
            label patchFacei = includedFaces_[patchi][i];

            pfld[patchFacei] = (Omega ^ (patchC[patchFacei] - origin_));
        }

        Ubf[patchi] == pfld;
    }
}
```

Firstly it is checked that the zone is active. Then the boundary field for `U` is obtained. After this the function loops over all elements in `includedFaces_`. Every element contains faces. The centres of these faces are found and the velocity is calculated as

$$\vec{u} = \vec{\Omega} \times (\vec{r}_f - \vec{r}_0)\,, \tag{2.3}$$

where $\vec{r_f}$ is the centre of the face, $\vec{r_0}$ the origin of rotation and $\vec{\Omega}$ the angular velocity. The calculated value then overrides the original boundary condition. This means that the `includedFaces_` are prescribed a solid body rotation. The absolute velocity is therefore fixed.

Returning to solving the velocity equation in `pimpleFoam` the second influence from MRF is in the term `MRF.DDt(U)`, added when solving the equation. This function is found in `MRFZoneList.C` and is shown below.

The function `DDt` in `MRFZoneList.C`

```
Foam::tmp<Foam::volVectorField> Foam::MRFZoneList::DDt
(
    const volVectorField& U
) const
{
    tmp<volVectorField> tacceleration
    (
        new volVectorField
        (
            IOobject
            (
                "MRFZoneList:acceleration",
                U.mesh().time().timeName(),
                U.mesh()
            ),
            U.mesh(),
            dimensionedVector(U.dimensions()/dimTime, Zero)
        )
    );
    volVectorField& acceleration = tacceleration.ref();

    forAll(*this, i)
    {
        operator[](i).addCoriolis(U, acceleration);
    }

    return tacceleration;
}
```

The function contains two main steps, constructing the acceleration vector field and, for every `MRFZone`, adding the Coriolis acceleration. For wheel modelling the interest is mainly in cases with a constant rotational velocity, hence the first acceleration will not be discussed in greater detail. Instead the function `addCoriolis` in `MRFZone` is called, which is shown below.

The function `addCoriolis` in `MRFZone.C`

```
void Foam::MRFZone::addCoriolis
(
    const volVectorField& U,
    volVectorField& ddtU
) const
{
    if (cellZoneID_ == -1)
    {
        return;
    }

    const labelList& cells = mesh_.cellZones()[cellZoneID_];
    vectorField& ddtUc = ddtU.primitiveFieldRef();
    const vectorField& Uc = U;

    const vector Omega = this->Omega();

    forAll(cells, i)
    {
        label celli = cells[i];
        ddtUc[celli] += (Omega ^ Uc[celli]);
    }
}
```

If the zone is active all cells in the zone is iterated over, adding a term to the acceleration according to $\vec{\Omega} \times \vec{u}$. The terms from all zones are then included in the solution of the velocity equation.

Next the contributions from MRF appears in `pEqn.H`. Firstly the function `MRF.zeroFilter` is called in the beginning of the pimple correction, which removes the contribution from the MRF when calculating `phiHbyA`. Next the function `MRF.makeRelative(phiHbyA)`, is called. This function is located in `MRFZoneList.C` and simply calls the `makeRelative(phi)` in `MRFZone.C` for every zone in the list, which in turn calls `makeRelativeRhoFlux(geometricOneField(), phi)` in `MRFZoneTemplates.C`. This function is shown below.

The first function named `makeRelativeRhoFlux` in `MRFZoneTemplates.C`

```
template<class RhoFieldType>
void Foam::MRFZone::makeRelativeRhoFlux
(
    const RhoFieldType& rho,
    surfaceScalarField& phi
) const
{
    if (!active_)
    {
        return;
    }

    const surfaceVectorField& Cf = mesh_.Cf();
    const surfaceVectorField& Sf = mesh_.Sf();

    const vector Omega = omega_->value(mesh_.time().timeOutputValue())*axis_;

    const vectorField& Cfi = Cf;
    const vectorField& Sfi = Sf;
    scalarField& phii = phi.primitiveFieldRef();

    // Internal faces
    forAll(internalFaces_, i)
    {
        label facei = internalFaces_[i];
        phii[facei] -= rho[facei]*(Omega ^ (Cfi[facei] - origin_)) & Sfi[facei];
    }

    makeRelativeRhoFlux(rho.boundaryField(), phi.boundaryFieldRef());
}
```

If the zone is active the function will loop over all internal faces and calculate the flux according to

$$\phi_{\text{rel}} = \phi_{\text{abs}} - \rho\left(\vec{\Omega} \times (\vec{r}_f - \vec{r}_0)\right) \cdot \vec{n}_f, \tag{2.4}$$

where $\vec{n}_f$ is the face vector. This means that the fluxes on the `internalFaces_` are the interpolated absolute velocity minus the rotation. Lastly the function `makeRelativeRhoFlux` (note that the name is the same but the parameter types are different, hence calling another function) is called. This function is also located in `MRFZoneTemplates.C` and is shown below.

The second function named `makeRelativeRhoFlux` in `MRFZoneTemplates.C`

```
template<class RhoFieldType>
void Foam::MRFZone::makeRelativeRhoFlux
(
    const RhoFieldType& rho,
    FieldField<fvsPatchField, scalar>& phi
) const
{
    if (!active_)
    {
        return;
    }

    const surfaceVectorField& Cf = mesh_.Cf();
    const surfaceVectorField& Sf = mesh_.Sf();

    const vector Omega = omega_->value(mesh_.time().timeOutputValue())*axis_;

    // Included patches
    forAll(includedFaces_, patchi)
    {
        forAll(includedFaces_[patchi], i)
        {
            label patchFacei = includedFaces_[patchi][i];

            phi[patchi][patchFacei] = 0.0;
        }
    }

    // Excluded patches
    forAll(excludedFaces_, patchi)
    {
        forAll(excludedFaces_[patchi], i)
        {
            label patchFacei = excludedFaces_[patchi][i];

            phi[patchi][patchFacei] -=
                rho[patchi][patchFacei]
              * (Omega ^ (Cf.boundaryField()[patchi][patchFacei] - origin_))
              & Sf.boundaryField()[patchi][patchFacei];
        }
    }
}
```

If the zone is active the function firstly treats the `includedFaces_`, where the relative flux is set to zero. Next the relative flux for the `excludedPatches_` is calculated. This is done the same way as for the `internalFaces_`, described in Eq. (2.4).

The last effect of MRF in `pimpleFoam` is when constraining the pressures were, similarly to the process described above, the relative fluxes are used in the calculation.

2.3 Rotating mesh

Since the aim of this method is to increase the understanding for how rotating meshes work, rather than understanding them in detail and later modify them, this explanation will contain less details than the description of MRF. Instead the focus will be on how the rotation is handled at a solver level.

Similarly to the description of MRF the start of `pimpleFoam` is examined (see code in Section 2.2.1). The implementation of dynamic mesh is noted at rows 78 and 100. At row 78 `#include "dynamicFvMesh.H"` can be found and on row 100 `#include "createDynamicFvMesh.H"`. At the latter the mesh is created, with the option for it being either static or dynamic. A dynamic mesh is selected if the dictionary `dynamicFvMesh` is found. Next the code updating the mesh is examined. This is found inside the `pimple` corrector loop and is shown below.

if-statement in the `pimple` corrector loop in `pimpleFoam.C`

```
127            if (pimple.firstIter() || moveMeshOuterCorrectors)
128            {
129                // Do any mesh changes
130                mesh.controlledUpdate();
```

```

                if (mesh.changing())
                {
                    MRF.update();

                    if (correctPhi)
                    {
                        // Calculate absolute flux
                        // from the mapped surface velocity
                        phi = mesh.Sf() & Uf();

                        #include "correctPhi.H"

                        // Make the flux relative to the mesh motion
                        fvc::makeRelative(phi, U);
                    }

                    if (checkMeshCourantNo)
                    {
                        #include "meshCourantNo.H"
                    }
                }
            }
```

Here it can be seen that the updating of the mesh occurs at the first iteration of the `pimple` loop. However, there is also an option for triggering the update at the outer correctors, which is controlled by `moveMeshOuterCorrectors`. When an update should be performed the function `controlledUpdate` is called and the mesh is moved. Next the solver checks if the mesh is changing. This code can be found in `$FOAM_SRC/OpenFOAM/meshes/polyMesh/polyMesh.H` and returns `true` if the mesh is moving or if the topology of the mesh is changing, as shown below.

The function `changing` in `polyMesh.H`

```
            //- Is mesh changing (topology changing and/or moving)
            bool changing() const
            {
                return moving()||topoChanging();
            }
```

In the case of, for example, a rotating rim the mesh would be moving, and hence the function would return `true`. If the mesh is changing any MRF zones are updated, as discussed briefly in Section 2.2. Next the absolute flux is calculated using the mapped surface velocity `Uf`. This field is created, if needed, when starting the solver by `#include "createUfIfPresent.H"`. The file `createUfIfPresent.H` can be found in

`$FOAM_SRC/finiteVolume/cfdTools/incompressible/createUfIfPresent.H`

and is (apart from the header) shown below.

`createUfIfPresent.H`

```
autoPtr<surfaceVectorField> Uf;

if (mesh.dynamic())
{
    Info<< "Constructing face velocity Uf\n" << endl;

    Uf.reset
    (
        new surfaceVectorField
        (
            IOobject
            (
                "Uf",
                runTime.timeName(),
                mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::AUTO_WRITE
            ),
            fvc::interpolate(U)
        )
    );
}
```

If the mesh is dynamic, which for this discussion it is, `mesh.dynamic()` will call the function in `dynamicFvMesh.H` which returns `true`. The interpolated face velocity is then read if a file is present, otherwise it is created. Next `phi` is corrected using the function `CorrectPhi` inside `correctPhi.H`, using the absolute flux just calculated. The flux is then again converted to be relative to the mesh motion using `fvc::makeRelative(phi, U)`, found in

`$FOAM_SRC/finiteVolume/finiteVolume/fvc/fvcMeshPhi.C`

and shown below.

The function `makeRelative` in `fvcMeshPhi.C`

```
void Foam::fvc::makeRelative
(
    surfaceScalarField& phi,
    const volVectorField& U
)
{
    if (phi.mesh().moving())
    {
        phi -= fvc::meshPhi(U);
    }
}
```

In the code it can be seen that the flux from the mesh is subtracted from the absolute flux. The function `meshPhi` is found in the same file and is shown below.

The function `meshPhi` in `fvcMeshPhi.C`

```
Foam::tmp<Foam::surfaceScalarField> Foam::fvc::meshPhi
(
    const volVectorField& vf
)
{
    return fv::ddtScheme<vector>::New
    (
        vf.mesh(),
        vf.mesh().ddtScheme("ddt(" + vf.name() + ')')
    ).ref().meshPhi(vf);
}
```

It can be seen that the flux returned depends on the numerical scheme used, which is to be expected. How this flux is obtained for the different schemes will not be investigated further here.

Examining the solver further relative flux is then used for calculating the velocity and pressure. Finally, at the end of `pEqn.H` two more lines of code directly related to the dynamic mesh can be found.

End of `pEqn.H` included in `pimpleFoam.C`

```
// Correct Uf if the mesh is moving
fvc::correctUf(Uf, U, phi);

// Make the fluxes relative to the mesh motion
fvc::makeRelative(phi, U);
```

Firstly, `Uf` is corrected. The code for this is found in `fvcMeshPhi.C` and is shown below.

The function `correctUf` in `fvcMeshPhi.C`

```
void Foam::fvc::correctUf
(
    autoPtr<surfaceVectorField>& Uf,
    const volVectorField& U,
    const surfaceScalarField& phi
)
{
    const fvMesh& mesh = U.mesh();

    if (mesh.dynamic())
    {
        Uf() = fvc::interpolate(U);
        surfaceVectorField n(mesh.Sf()/mesh.magSf());
        Uf() += n*(phi/mesh.magSf() - (n & Uf()));
    }
}
```

If the mesh is dynamic the velocity `U` is interpolated to the faces of the cells. The resulting velocity is then corrected by adding a factor

$$\frac{\vec{n}}{|\vec{n}|}\left(\frac{\phi}{|\vec{n}|} - (\vec{n}\cdot\vec{u}_f)\right), \tag{2.5}$$

where $\vec{n}$ is the normal of the cell face, $|\vec{n}|$ the face area, ϕ the flux and $\vec{u}_f$ the interpolated face velocity. This correction ensures that the face-normal component of $\vec{u}_f$ is the same as the velocity corresponding to the flux ϕ. Returning to the solver the final step is to calculate the relative flux which is done using the same function as described above.

Chapter 3

Modifications to MRF

At the contact patch the grooves move parallel to the ground rather than along a circular path. However, if MRF is used in the grooves a rotation will be added to the flow which, around the contact patch, then might be unrepresentative. Therefore, in this section the process of modifying the MRF method to constrain how the velocity is set in certain cells will be presented.

3.1 Creating own copies of solvers and libraries

Before being able to modify the MRF method it is necessary to create a copy of the library in our own workspace. Since the MRF is implemented in `finiteVolume` a modified version of this will be created. Next a solver will be modified to use the new MRF method. For this, `simpleFoam` will be used. Firstly the MRF implementation in `finiteVolume` is copied and renamed. Opening a new terminal window the following commands are executed:

```
OFv2006 # Source the OpenFOAM installation
cd $WM_PROJECT_USER_DIR
mkdir -p src/finiteVolume/cfdTools/general
cp -r $FOAM_SRC/finiteVolume/cfdTools/general/MRF src/finiteVolume/cfdTools/general/
cd src/finiteVolume/cfdTools/general/MRF
mv IOMRFZoneList.C myIOMRFZoneList.C
mv IOMRFZoneList.H myIOMRFZoneList.H
mv MRFZone.C myMRFZone.C
mv MRFZone.H myMRFZone.H
mv MRFZoneList.C myMRFZoneList.C
mv MRFZoneList.H myMRFZoneList.H
mv MRFZoneListTemplates.C myMRFZoneListTemplates.C
mv MRFZoneI.H myMRFZoneI.H
mv MRFZoneTemplates.C myMRFZoneTemplates.C
sed -i 's/MRFZone/myMRFZone/g' my* # Rename classes
sed -i 's/IOmy/myIO/g' myIOMRFZoneList.* # Fix names containing IO
```

Next create the `Make` directory containing `files` and `options`.

```
cd $WM_PROJECT_USER_DIR/src/finiteVolume
mkdir Make
touch Make/files Make/options
```

In `Make/files` the following is added

`Make/files` in `myFiniteVolume`

```
cfdTools/general/MRF/myIOMRFZoneList.C
cfdTools/general/MRF/myMRFZone.C
```

```
cfdTools/general/MRF/myMRFZoneList.C

LIB = $(FOAM_USER_LIBBIN)/libmyFiniteVolume
```

and in `Make/options`

`Make/options` in `myFiniteVolume`

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools
```

Then the library can be compiled and links to the libraries can be created.

```
wmake
wmakeLnInclude .
```

Next we will modify a solver for implementing the new MRF model in. This will be done for `simpleFoam`. The solver is copied and renamed to `mySimpleFoam`. Additionally the parts of `simpleFoam` that will not be used are removed.

```
cd $WM_PROJECT_USER_DIR
mkdir -p applications/solvers/incompressible
cp -r $FOAM_APP/solvers/incompressible/simpleFoam applications/solvers/incompressible/
cd applications/solvers/incompressible
mv simpleFoam mySimpleFoam
cd mySimpleFoam
rm -r overSimpleFoam porousSimpleFoam SRFSimpleFoam
mv simpleFoam.C mySimpleFoam.C
sed -i 's/simpleFoam/mySimpleFoam/g' mySimpleFoam.C
```

Next `Make/files` is modified to

`Make/files` in `mySimpleFoam`

```
mySimpleFoam.C

EXE = $(FOAM_USER_APPBIN)/mySimpleFoam
```

In `Make/options` we need to add information on where to find the new version of `finiteVolume`. Therefore `Make/options` is modified to

`Make/options` in `mySimpleFoam`

```
LIB_USER_SRC = $(WM_PROJECT_USER_DIR)/src

EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_USER_SRC)/finiteVolume/lnInclude

EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -lincompressibleTransportModels \
    -latmosphericModels \
    -lmyFiniteVolume
```

where the lines 1, 11, 14 and 23 has been added. The solver can then be compiled with `wmake`.

3.2 Modifying the solver

After successfully compiling `mySimpleFoam`, which as of now has not been modified compared to the original, the changes can be made so it uses the MRF functionalities from `myFiniteVolume` instead of `finiteVolume`. In Section 2.2 it was described how MRF was implemented in `pimpleFoam`. For `simpleFoam` the implementation is similar. At the end of `createFields.H` the file `createMRF.H` is included. As explained in Section 2.2 this file contains a single line: `IOMRFZoneList MRF(mesh);`. In order to use the new MRF library the line `#include "createMRF.H"` is changed to `myIOMRFZoneList MRF(mesh);` in `createFields.H`. Since this object is of the type `myIOMRFZoneList` the new MRF method will now be used. In order for the solver to know about the class the header file also needs to be included in `mySimpleFoam`. This is added together with the other libraries outside `main` as

Included libraries in `mySimpleFoam.C`

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "simpleControl.H"
#include "fvOptions.H"
#include "myIOMRFZoneList.H"
```

The solver can now be recompiled with `wmake`.

3.3 Modifying the MRF method

The aim of the modifications is to change how the MRF method sets the velocity in cells close to the contact patch. Since the tyre is deformed the grooves travel parallel to the ground rather than along a circular path when close to the ground. For a realistic contact patch the effect of this change is likely negligible. However, it can be done as a useful exercise to see how the MRF method can be modified.

The determination of which cells should follow the circular path and which should follow the linear path could be done in multiple ways. Here this will be done by specifying an additional input in `MRFProperties`, namely a value for the z-coordinate. For cells included in the MRF zone with a z-coordinate smaller than this value the linear path will be used. Cells with a z-coordinate larger than this value will be treated as in the usual MRF method.

Next a method is needed for calculating the velocity that should be applied to the cells instead of the rotation. In order to add as little as possible to the code it is desirable to only use what is already specified in `MRFProperties`, namely the centre of rotation and the rotational velocity. One method to accomplish the desired velocity could be to set a specific velocity along a specific axis. However, this solution would not be very general, failing for example if the toe angle (rotation around the vector normal to the ground) of the tyre is changed. Instead the method for calculating the velocity will initially be kept as is. Afterwards, if the current cell is below the specified z-coordinate, the z-component of the velocity will be set to zero, constraining the velocity to the xy-plane.

Firstly the MRF method will be modified to handle the new options in `MRFProperties`. In addition to the z-coordinate, below which the modification will be active, an option for deactivating the modification will also be added. This is done by firstly moving to the directory of the copied library.

```
cd $WM_PROJECT_USER_DIR/src/finiteVolume/cfdTools/general/MRF
```

Next the new properties can be added to the declaration of the class in `myMRFZone.H`. These are added in the section for `private data`, right after `omega_`

Part of private data in `myMRFZone.H`

```
        // Should the cutoff function be used?
        bool useCutoff_;

        // Cutoff for z
        autoPtr<Function1<scalar>> zCutoff_;
```

Secondly a function will be needed for obtaining the value of `zCutoff_`. Therefore the following is added under `member functions` right after the function `Omega`.

Part of member functions in `myMRFZone.H`

```
        // Return the current value for z cutoff
        double zCutoff() const;
```

Next the constructor in `myMRFZone.C` is modified. The beginning of the modified constructor is shown below.

Beginning of the constructor in `myMRFZone.C`

```
239 Foam::myMRFZone::myMRFZone
240 (
241     const word& name,
242     const fvMesh& mesh,
243     const dictionary& dict,
244     const word& cellZoneName
245 )
246 :
247     mesh_(mesh),
248     name_(name),
249     coeffs_(dict),
250     active_(coeffs_.getOrDefault("active", true)),
251     cellZoneName_(cellZoneName),
252     cellZoneID_(),
253     excludedPatchNames_
254     (
255         coeffs_.getOrDefault<wordRes>("nonRotatingPatches", wordRes())
256     ),
257     origin_(coeffs_.get<vector>("origin")),
258     axis_(coeffs_.get<vector>("axis").normalise()),
259     omega_(Function1<scalar>::New("omega", coeffs_)),
260     useCutoff_(coeffs_.getOrDefault("useCutoff", false)),
261     zCutoff_()
262 {
263     if (cellZoneName_ == word::null)
264     {
265         coeffs_.readEntry("cellZone", cellZoneName_);
266     }
267 
268     if (useCutoff_)
269     {
270         Info<< "Using cutoff in MRF zone " << name_ << endl;
271         zCutoff_ = Function1<scalar>::New("zCutoff", coeffs_);
272     }
```

Firstly the new variables are added at lines 260 and 261. The value for `useCutoff_` is read from the dictionary. If this fails it defaults to `false`. The next addition is the `if`-statement at rows 268-272. There, if the cutoff should be used, this is stated in the output and the value is read from the dictionary.

Next, the function for obtaining the value of `zCutoff_` is added right after the function `Omega` as shown below.

zCutoff in myMRFZone.C

```
319 double Foam::myMRFZone::zCutoff() const
320 {
321     return zCutoff_->value(mesh_.time().timeOutputValue());
322 }
```

The code now needs to be modified at the locations were the velocity or fluxes are calculated. Similarly to `pimpleFoam`, `simpleFoam` calls the function `correctBoundaryVelocity` in `myMRFZone.C`. The function is modified as shown below.

correctBoundaryVelocity in myMRFZone.C

```
569 void Foam::myMRFZone::correctBoundaryVelocity(volVectorField& U) const
570 {
571     if (!active_)
572     {
573         return;
574     }
575 
576     const vector Omega = this->Omega();
577     const double zCutoff = this->zCutoff();
578 
579     // Included patches
580     volVectorField::Boundary& Ubf = U.boundaryFieldRef();
581 
582     forAll(includedFaces_, patchi)
583     {
584         const vectorField& patchC = mesh_.Cf().boundaryField()[patchi];
585 
586         vectorField pfld(Ubf[patchi]);
587 
588         forAll(includedFaces_[patchi], i)
589         {
590             label patchFacei = includedFaces_[patchi][i];
591 
592             pfld[patchFacei] = (Omega ^ (patchC[patchFacei] - origin_));
593 
594             if (useCutoff_) {
595                 if (patchC[patchFacei].component(2) < zCutoff)
596                 {
597                     pfld[patchFacei].component(2) = 0;
598                 }
599             }
600         }
601 
602         Ubf[patchi] == pfld;
603     }
604 }
```

First the value of the z-coordinate below which the modification should take place is obtained (row 577). Next the velocities are calculated just as before by looping over the faces. However, if the cutoff is used and if the z-coordinate of the centre of the face is below the specified cutoff coordinate the z-component of the calculated velocity is set to zero, as seen at rows 594-600.

The next term that needs to be adjusted is when adding the Coriolis acceleration, which is done in `addCoriolis` in `myMRFZone.C`. The function is modified to only add the Coriolis acceleration to the cells above the cutoff coordinate. This is shown below where the coordinate is obtained at row 341 and the `if`- and `else`-statement at rows 349-358 adds the Coriolis contribution to the desired cells.

addCoriolis in myMRFZone.C

```
325 void Foam::myMRFZone::addCoriolis
326 (
327     const volVectorField& U,
328     volVectorField& ddtU
) const
{
    if (cellZoneID_ == -1)
    {
        return;
    }

    const labelList& cells = mesh_.cellZones()[cellZoneID_];
    vectorField& ddtUc = ddtU.primitiveFieldRef();
    const vectorField& Uc = U;

    const vector Omega = this->Omega();
    const double zCutoff = this->zCutoff();

    const volVectorField& C = mesh_.C();

    forAll(cells, i)
    {
        label celli = cells[i];

        if (useCutoff_) {
            if (C[celli].component(2) > zCutoff)
            {
                ddtUc[celli] += (Omega ^ Uc[celli]);
            }
        }
        else
        {
            ddtUc[celli] += (Omega ^ Uc[celli]);
        }
    }
}
```

The final modifications needed are in the functions returning the relative fluxes. These can be found in `myMRFZoneTemplates.C` and the two functions `makeRelativeRhoFlux`. The first of these, handling the `internal faces` is shown below.

The first `makeRelativeRhoFlux` in `myMRFZoneTemplates.C`

```
template<class RhoFieldType>
void Foam::myMRFZone::makeRelativeRhoFlux
(
    const RhoFieldType& rho,
    surfaceScalarField& phi
) const
{
    if (!active_)
    {
        return;
    }

    const surfaceVectorField& Cf = mesh_.Cf();
    const surfaceVectorField& Sf = mesh_.Sf();

    const vector Omega = omega_->value(mesh_.time().timeOutputValue())*axis_;
    const double zCutoff = zCutoff_->value(mesh_.time().timeOutputValue());

    const vectorField& Cfi = Cf;
    const vectorField& Sfi = Sf;
    scalarField& phii = phi.primitiveFieldRef();

    // Internal faces
    forAll(internalFaces_, i)
    {
        label facei = internalFaces_[i];
        vector SfiMod = Sfi[facei];

        if (useCutoff_) {
```

```
 65             if (Cfi[facei].component(2) < zCutoff)
 66             {
 67                 SfiMod.component(2) = 0;
 68             }
 69         }
 70         phii[facei] -= rho[facei]*(Omega ^ (Cfi[facei] - origin_)) & SfiMod;
 71     }
 72 
 73     makeRelativeRhoFlux(rho.boundaryField(), phi.boundaryFieldRef());
 74 }
```

The first modification is found at row 52 where the cutoff coordinate is obtained. In the loop iterating over the faces the removal of the z-component has been achieved by setting the z-component of `Sf` to zero. This is equivalent to first removing the z-component from $\Omega \times (\vec{r}_f - \vec{r_0})$ and then calculating the scalar product. However, the chosen implementation saves some lines of code.

Next the modifcations are made to the function `makeRelativeRhoFlux`, handling `included patches` and `excluded patches`. This is done the same way as for the `internal faces`. The resulting code is shown below, where the modifications can be found at rows 93 and 112-119.

The second `makeRelativeRhoFlux` in `myMRFZoneTemplates.C`

```
 77 template<class RhoFieldType>
 78 void Foam::myMRFZone::makeRelativeRhoFlux
 79 (
 80     const RhoFieldType& rho,
 81     FieldField<fvsPatchField, scalar>& phi
 82 ) const
 83 {
 84     if (!active_)
 85     {
 86         return;
 87     }
 88 
 89     const surfaceVectorField& Cf = mesh_.Cf();
 90     const surfaceVectorField& Sf = mesh_.Sf();
 91 
 92     const vector Omega = omega_->value(mesh_.time().timeOutputValue())*axis_;
 93     const double zCutoff = zCutoff_->value(mesh_.time().timeOutputValue());
 94 
 95     // Included patches
 96     forAll(includedFaces_, patchi)
 97     {
 98         forAll(includedFaces_[patchi], i)
 99         {
100             label patchFacei = includedFaces_[patchi][i];
101 
102             phi[patchi][patchFacei] = 0.0;
103         }
104     }
105 
106     // Excluded patches
107     forAll(excludedFaces_, patchi)
108     {
109         forAll(excludedFaces_[patchi], i)
110         {
111             label patchFacei = excludedFaces_[patchi][i];
112             vector SfMod = Sf.boundaryField()[patchi][patchFacei];
113 
114             if (useCutoff_) {
115                 if (Cf.boundaryField()[patchi][patchFacei].component(2) < zCutoff)
116                 {
117                     SfMod.component(2) = 0;
118                 }
119             }
120 
121             phi[patchi][patchFacei] -=
122                 rho[patchi][patchFacei]
```

```
                * (Omega ^ (Cf.boundaryField()[patchi][patchFacei] - origin_))
                & SfMod;
        }
    }
}
```

Now all modifications have been implemented and the code can be recompiled with `wmake`.

Chapter 4

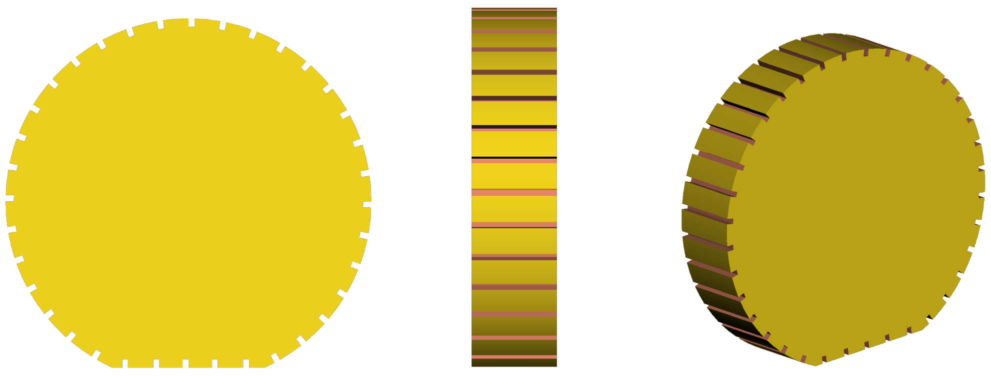
Test case

In the following test case a simplified tyre will be used to illustrate the differences between some of the different wheel rotation methods. The test case aims to show how to use the different models. Since the case should be fast to run the mesh is coarse. All files, including the geometry, needed to run the case can be found in the accompanying files.

The setup of the case is largely based on the motorbike tutorial found in

`$FOAM_TUTORIALS/incompressible/simpleFoam/motorBike`

The geometry, which was constructed in ANSA and exported in stl-format is shown in Figure 4.1. The tyre features lateral grooves in the tread as well as a flattened section next to the ground, which crudely mimics the contact patch. Note that the geometry is constructed such that the surfaces inside the grooves, named `tyre_grooves`, can be separated from the rest of the tyre, named `tyre`. This is illustrated by different colours in Figure 4.1. The tyre is constructed such that the rotational centre is at $(0, 0, 0)$ and the rotational axis is aligned with the y-axis. At the non-flattened part the tyre has a radius of 220 mm and the distance from the centre to the ground is 200 mm.



(a) Side view (b) Front view (c) Iso view

Figure 4.1: The simplified tyre used in the test case.

4.1 Rotating wall

The baseline case will be set up using the rotating wall boundary condition. Later this case will be modified to include the MRF method.

Below the directory tree for the case is shown. Here the files most relevant for the wheel rotation modelling will be presented.

```
rotating_wall
├── 0.orig
│   ├── include
│   │   ├── fixedInlet
│   │   ├── frontBackUpperPatches
│   │   └── initialConditions
│   ├── k
│   ├── nut
│   ├── omega
│   ├── p
│   └── U
├── constant
│   ├── transportProperties
│   ├── triSurface
│   │   └── tyre_grooves_flat.stl
│   └── turbulenceProperties
└── system
    ├── blockMeshDict
    ├── controlDict
    ├── decomposeParDict
    ├── fvSchemes
    ├── fvSolution
    ├── snappyHexMeshDict
    └── surfaceFeatureExtractDict
```

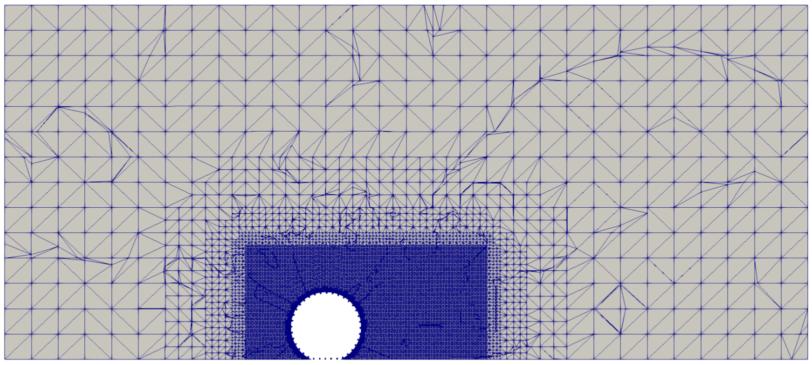
For meshing the OpenFOAM tool `snappyHexMesh` is used. However, firstly the computational domain is constructed using `blockMesh`. The definition of the vertices and the block is shown below. The entire `blockMeshDict` can be found in Appendix B.1.

Definition of `vertices` and `blocks` in `blockMeshDict`

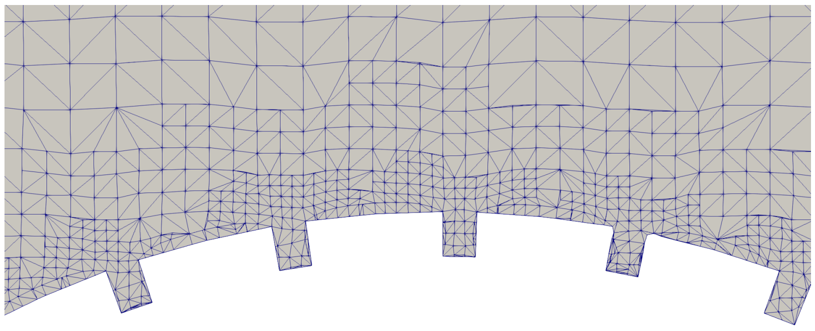
```
vertices
(
    (-2 -1 -0.199)
    (3 -1 -0.199)
    (3  1 -0.199)
    (-2  1 -0.199)
    (-2 -1 2)
    (3 -1 2)
    (3  1 2)
    (-2  1 2)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (30 12 14) simpleGrading (1 1 1)
);
```

It can be seen that a computational domain of $5 \times 2 \times 2.199$m is constructed and later meshed with $30 \times 12 \times 14$ cells. The reason for placing the ground at $z = -0.199$ m is to get a clean cut between the contact patch of the tyre and the ground.

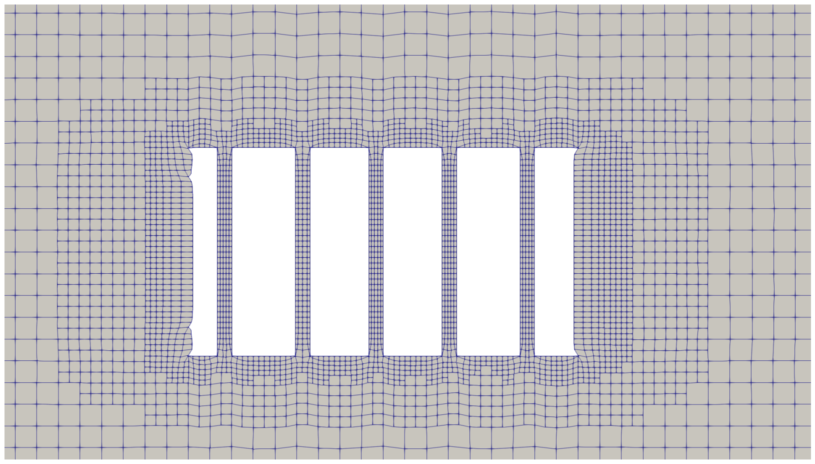
Next `surfaceFeatureExtraxt` and `snappyHexMesh` is used to obtain the actual mesh. Since this is not supposed to be a tutorial in meshing no details will be given for this process. For more details on the topic see for example the user guide by CFD Direct [6]. The dictionary files can be found in Appendix B.2 and B.3. The tyre geometry file, named `tyre_grooves_flat.stl`, is placed in `constant/triSurfaces`, as seen in the directory tree above. Figure 4.2 shows the mesh in the plane $y = 0$ and at the contact patch. Figure 4.2a shows the entire domain and Figure 4.2b shows the mesh in the grooves on top of the tyre, both at $y = 0$. In Figure 4.2c the mesh at the ground around the contact patch is shown and the imprint from the lateral grooves on the ground can be seen.



(a) Cross section of mesh in domain at $y = 0$.



(b) Close up of mesh in grooves at top of tyre at $y = 0$.



(c) Mesh at the ground around the contact patch.

Figure 4.2: Mesh at $y = 0$ and the contact patch. Flow is from left to right.

The method for setting the boundary conditions is inspired by the **motorbike** tutorial, where files in `0.orig/include` is used for values used by multiple quantities. The active lines of these files are shown below.

`initialConditions`

```
flowVelocity         (38.89 0 0);
pressure             0;
turbulentKE          0.24;
turbulentOmega       1.78;
```

`fixedInlet`

```
inlet
{
    type  fixedValue;
    value $internalField;
}
```

`frontBackUpperPatches`

```
upperWall
{
    type slip;
}

frontAndBack
{
    type slip;
}
```

Comparing to the `motorbike` tutorial the only significant change is that the variable `flowVelocity` in `initialConditions` has been altered to $(38.89, 0, 0)$ m/s, corresponding to 140 km/h. These files are then used in the definition of boundary conditions for `U`, which is shown below.

Boundary conditions for velocity in `U`

```
#include        "include/initialConditions"

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform $flowVelocity;

boundaryField
{
    #includeEtc "caseDicts/setConstraintTypes"

    #include "include/fixedInlet"

    outlet
    {
        type            zeroGradient;
        inletValue      uniform (0 0 0);
        value           $internalField;
    }

    lowerWall
    {
        type            fixedValue;
        value           $internalField;
    }

    "(tyre|tyre_grooves)"
    {
        type            rotatingWallVelocity;
        origin          (0 0 0);
        axis            (0 1 0);
        omega           -194.45;
    }

    #include "include/frontBackUpperPatches"
}
```

Here it can be seen that the `rotatingWallVelocity` boundary condition has been used for `tyre` and `tyre_grooves`. The origin and axis of rotation has been specified, along with the rotational velocity which has been calculated as

$$\omega = \frac{v}{r} = \frac{38.89\,\mathrm{m/s}}{0.2\,\mathrm{m}} = 194.45\,\mathrm{s}^{-1}. \tag{4.1}$$

Furthermore a moving wall boundary condition is used on the `lowerWall` (ground). For the remaining fields `k`, `nut`, `omega` and `p` the boundary conditions are the same as in the `motorbike` tutorial. The only modification needed is to change the name of the boundary from `motorBikeGroup` to `"(tyre|tyre_grooves)"` since the surfaces of the tyre is treated the same way as the surfaces of the motorbike for these quantities.

The case is then solved using `simpleFoam`. All steps can be executed using the `Allrun` script, which is shown below.

`Allrun`

```
#!/bin/sh
cd "${0%/*}" || exit                                # Run from this directory
. ${WM_PROJECT_DIR:?}/bin/tools/RunFunctions        # Tutorial run functions
#------------------------------------------------------------------------------

decompDict="-decomposeParDict system/decomposeParDict"

runApplication surfaceFeatureExtract
runApplication blockMesh
runApplication $decompDict decomposePar
runParallel $decompDict snappyHexMesh -overwrite
restore0Dir -processor
runParallel $decompDict $(getApplication)
runApplication reconstructParMesh -constant
runApplication reconstructPar

#------------------------------------------------------------------------------
```

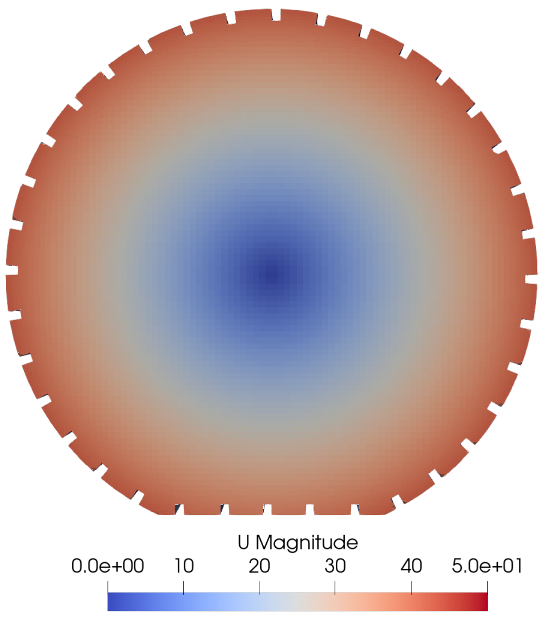
In the script `snappyHexMesh` and `simpleFoam` is run in parallel, hence the steps of decomposing and reconstructing the mesh and solution has been added.

In Figure 4.3 the magnitude and x-component of velocity is shown at the tyre surface. It can be seen that, for the surface shown, a representative boundary condition is obtained. The magnitude of the velocity is dependant on the distance from the rotational axis and the velocity is applied such that the x-component is negative at the top of the tyre and positive at the lower half. In Figure 4.4 the velocity magnitude in the plane $y = 0$ is shown. Here the shortcomings of the rotating wall boundary condition can be seen, where it fails to reproduce a realistic velocity in the lateral grooves.

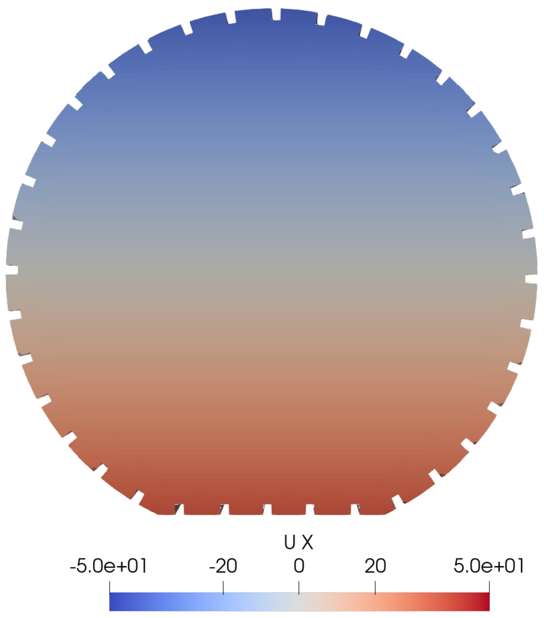
4.2 MRF

In order to add a representative velocity boundary condition in the lateral grooves of the tyre the MRFg approach, where MRF zones are added in the lateral grooves, proposed by Hobeika and Sebben [4] will be used.

The same setup as for rotating wall is used. If desired the velocity boundary condition for the grooves could be altered to be stationary instead of rotating since the rotation now will be achieved by the MRF. However, as seen in Section 2.2, MRF will override the boundary conditions. Next the dictionary `constant/MRFProperties` is created, with the following content.



(a) Velocity magnitude at the tyre surface.



(b) x-component of velocity at the tyre surface.

Figure 4.3: Velocity at the tyre using the rotating wall boundary condition.

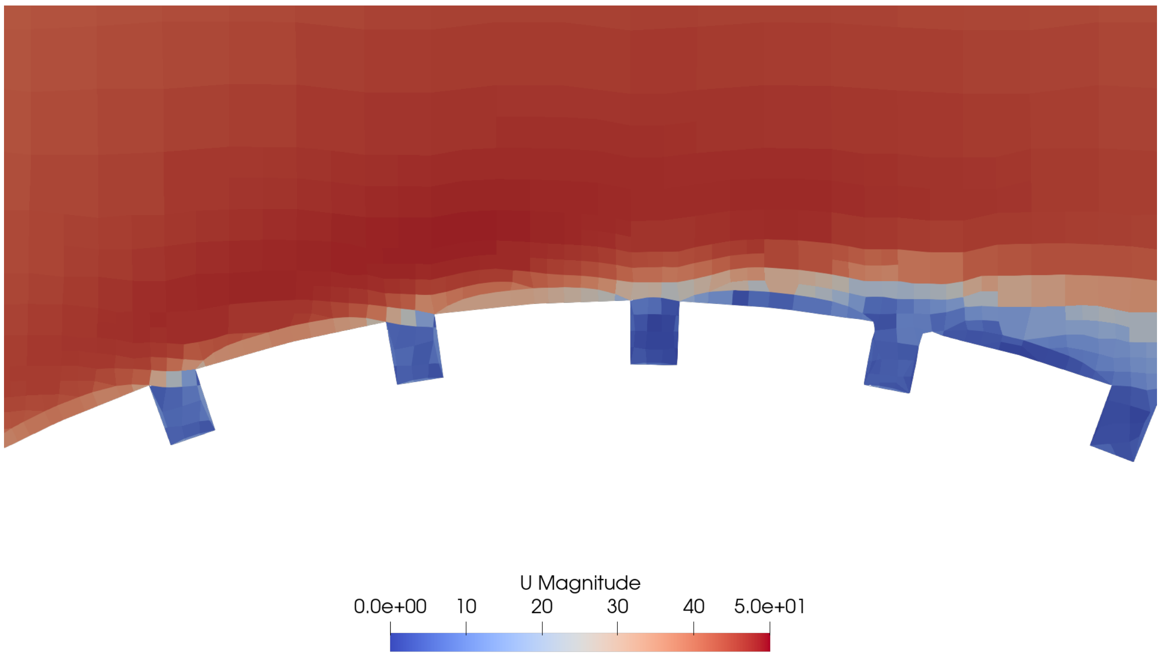


Figure 4.4: Velocity magnitude in $y = 0$ at the top of the tyre using rotating wall.

MRFProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      MRFProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

MRF1
{
    cellZone    MRFZoneSHM;
    active      yes;

    // Fixed patches (by default they 'move' with the MRF zone)
    nonRotatingPatches ();

    origin          (0 0 0);
    axis            (0 1 0);
    omega           -194.45;
}

// ************************************************************************* //
```

Here one MRF zone named `MRF1` is defined. The cells in the `cellZone` `MRFZoneSHM` are added and the zone is set to be active. Furthermore no patches are excluded and the origin, axis and angular velocity is specified, similarly to the rotating wall boundary condition. Before meshing the cell zone needs to be specified. This is done in `snappyHexMesh` by defining a cylinder that coincides with the outside of the tyre, meaning that the grooves will be inside the cylinder. This is done by adding the following to the `geometry` section of `snappyHexMeshDict`

Definition of cylinder in `snappyHexMeshDict`

```
    MRFZoneSHM
    {
        type searchableCylinder;
        point1 (0.000 -0.050 0.000);
        point2 (0.000  0.050 0.000);
        radius 0.220;
    }
```

Later, under `refinementSurfaces`, the following is added, specifying that the cells inside the cylinder should be added to the `cellZone` named `MRFZoneSHM`.

Definition of `cellZone` in `snappyHexMeshDict`

```
        MRFZoneSHM
        {
            level (5 5);
            cellZone MRFZoneSHM;
            faceZone MRFZoneSHMf;
            cellZoneInside inside;
        }
```

Now the case can be meshed and solved just as for the rotating wall case. In the output it can be seen that `MRFProperties` is read and one MRF zone is created.

```
Creating MRF zone list from MRFProperties
    creating MRF zone: MRF1
```

The results shows that the surface velocity outside of the grooves are the same as for rotating wall, which is to be expected. However, comparing the velocity magnitude in the grooves, which is shown in Figure 4.5, it can be seen that there now is a velocity source present in the grooves because of the added MRF zone.

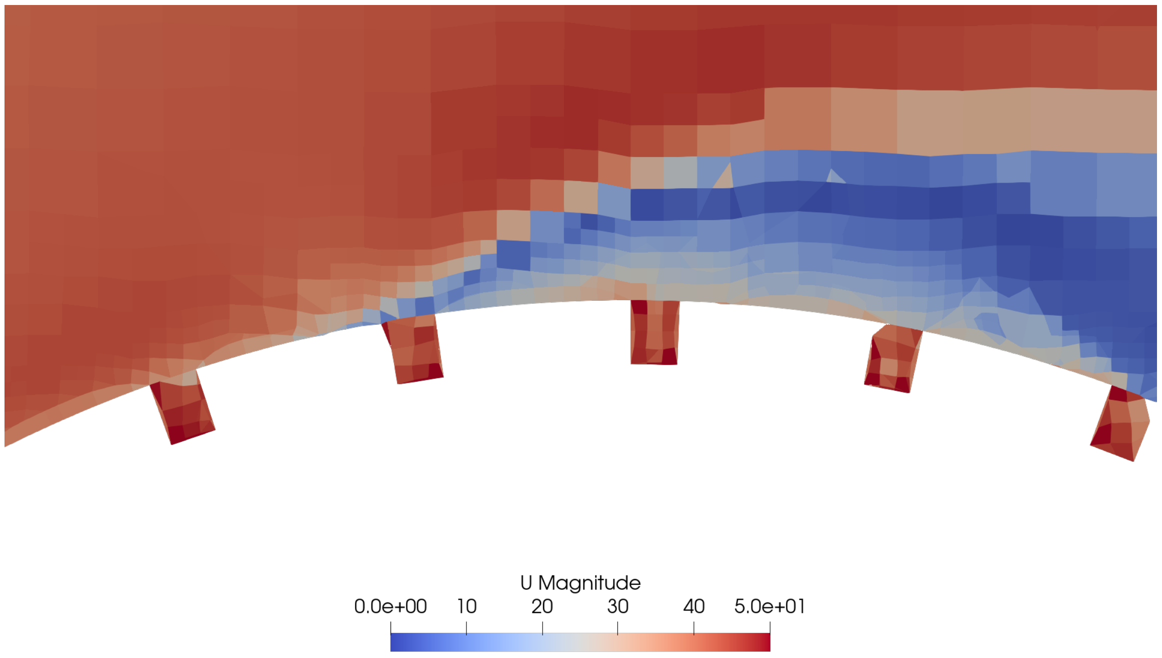


Figure 4.5: Velocity magnitude in $y = 0$ at the top of the tyre using MRFg.

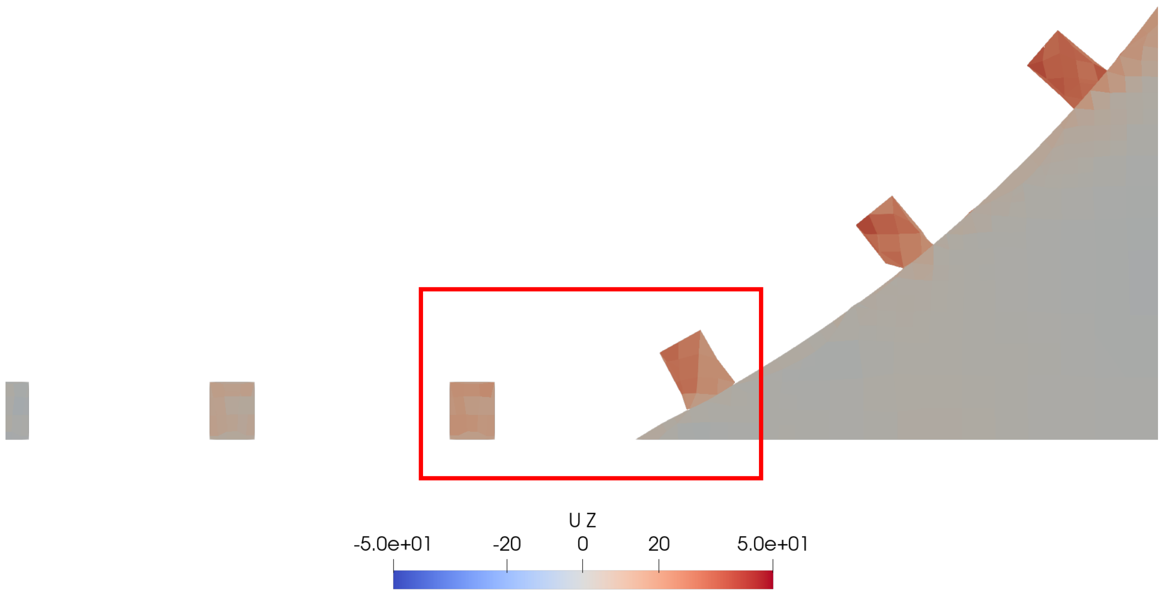
4.3 Modified MRF

In order to use the modified MRF method two changes are needed. Firstly the `application` in `controlDict` is changed to `mySimpleFoam`. Secondly the options for activating the cutoff and the z-coordinate are specified in `MRFProperties`, which is shown below.

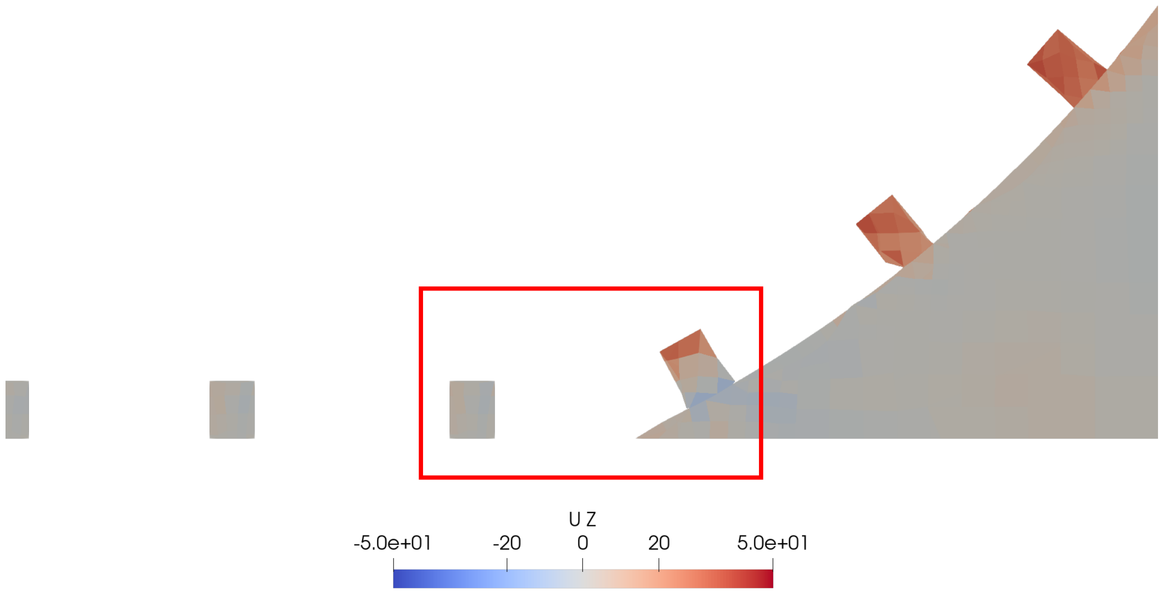
`MRFProperties` using the modified MRF method

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      MRFProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

MRF1
{
    cellZone    MRFZoneSHM;
    active      yes;

    // Fixed patches (by default they 'move' with the MRF zone)
    nonRotatingPatches ();

    origin        (0 0 0);
    axis          (0 1 0);
    omega         -194.45;
    useCutoff     yes;
    zCutoff       -0.189;
}

// ************************************************************************* //
```

In Figure 4.6 the resulting velocity in the z-direction is shown at the rear edge of the contact patch in the plane $y = 0$. It can be seen that the modified MRF method clearly reduces the z-component of the velocity, removing the undesired velocity component. This is most visible for the rearmost groove in the contact patch and the first groove downstream the contact patch, highlighted in Figure 4.6. Furthermore Figure 4.7 shows the velocity magnitude in the same plane at the top of the tyre. In this region the modifications should not have any effect. Apart from minor differences, which can be explained by the simulations not being fully converged, it can be seen that the original and modified MRF method produces the same results away from the modified region, which is desired.

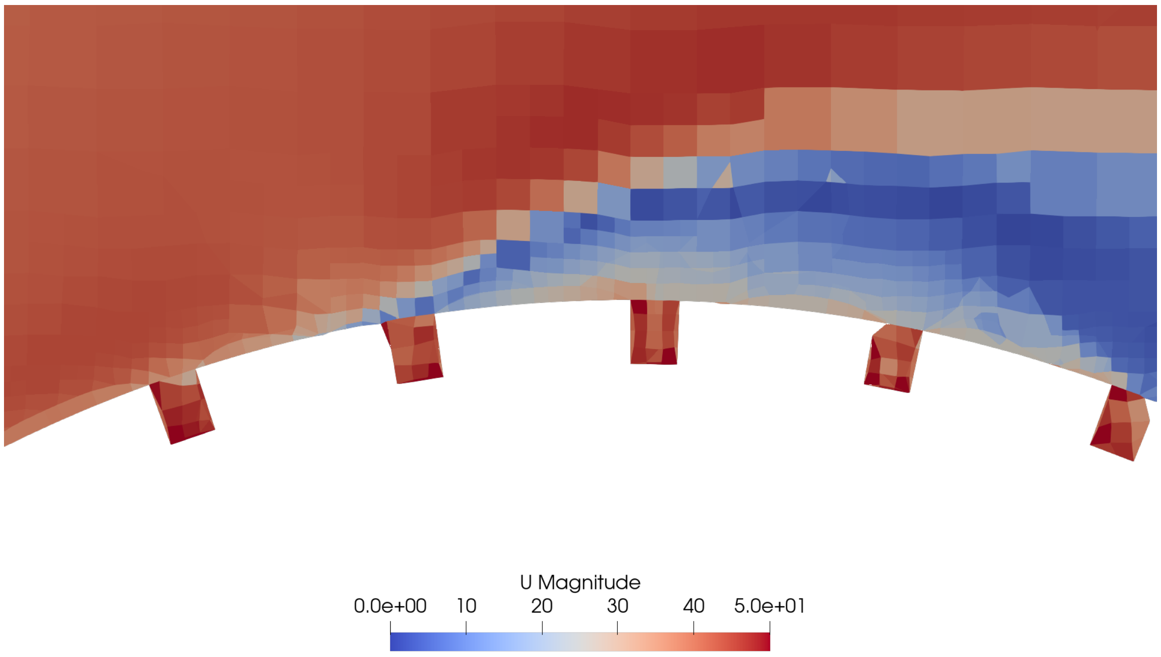


(a) Original MRF method

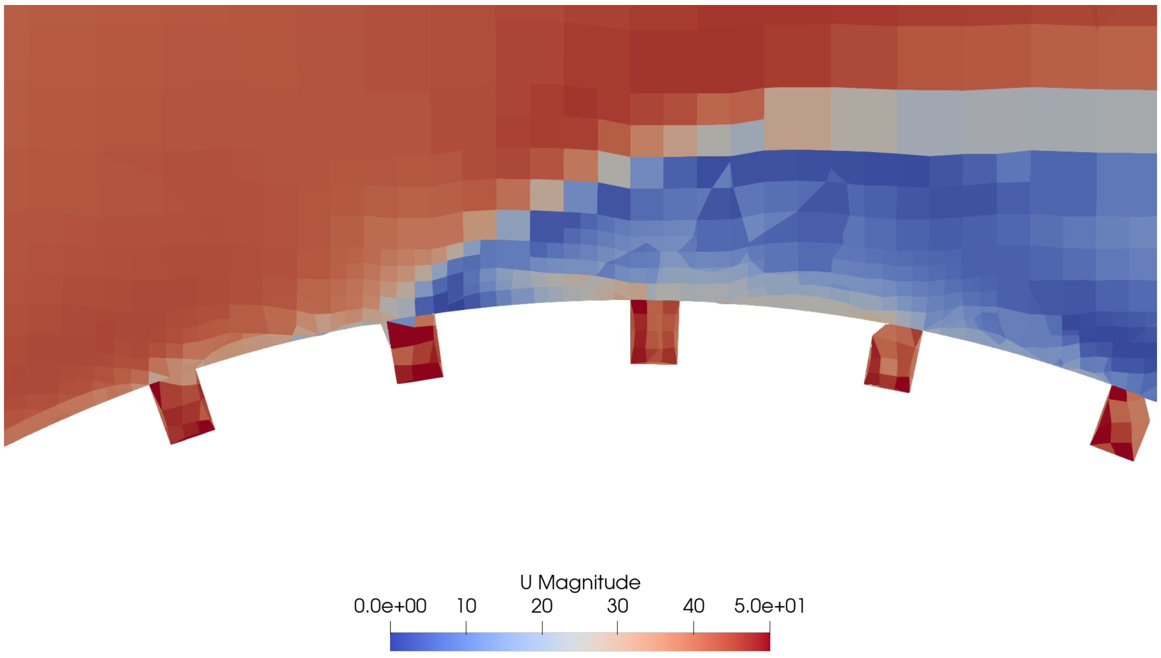


(b) Modified MRF method

Figure 4.6: Velocity in z-direction at the rear edge of the contact patch in $y = 0$.



(a) Original MRF method



(b) Modified MRF method

Figure 4.7: Velocity magnitude at the top of the tyre in $y = 0$.

Chapter 5

Conclusions

In this tutorial an introduction to rotation methods for wheel aerodynamic simulations was given. The methods rotating wall, multiple reference frame (MRF) and rotating mesh were presented and the corresponding benefits and drawbacks were discussed. Additionally, the possibility of combining the methods, creating hybrid methods, was presented.

Next, an explanation of the implementation of the different methods in OpenFOAM was given, with an extra focus on MRF. Important aspects of the code were presented and it was shown how the execution of the code was altered by the different methods.

Having explained the implementation of the methods an example of how to modify the MRF method was given, where a cutoff coordinate was introduced to avoid adding momentum in the vertical direction for cells near the contact patch of a tyre.

Finally, an example of how to use the MRF method for modelling the lateral grooves of a tyre was given. The results were compared to a baseline case using only the rotating wall boundary condition. Next the original MRF implementation was compared to the modified version, were it was illustrated how the flow was altered below the specified cutoff coordinate.

Bibliography

[1] "See the MRF development." http://openfoamwiki.net/index.php/See_the_MRF_development/. Accessed: 2020-12-15.

[2] C. Landström, S. Sebben, and L. Löfdahl, "Effects of wheel orientation on predicted flow field and forces when modelling rotating wheels using cfd," in *8th MIRA international vehicle aerodynamics conference*, 2010.

[3] T. Hobeika, *Wheel Modelling and Cooling Flow Effects on Car Aerodynamics.* PhD thesis, Chalmers University of Technology, Gothenburg, 2018.

[4] T. Hobeika and S. Sebben, "CFD investigation on wheel rotation modelling," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 174, pp. 241–251, Mar. 2018.

[5] H. Nilsson, "Rotating machinery training at OFW10," June 2015.

[6] C. Greenshields, "OpenFOAM v8 User Guide: 5.4 Meshing with snappyHexMesh," July 2020. Section: User Guide.

Study questions

How to use it:

1. Is it important to specify a normalised rotational vector for the rotating wall boundary condition? Will the length of the vector affect the rotation?
2. Where is the code for MRF found?

The theory of it:

1. What is the main drawback of the rotating wall boundary condition when modelling realistic tyres?
2. List some of the benefits and drawbacks of different boundary conditions used for modelling wheels.

How it is implemented:

1. In which file can it be seen how the Coriolis contribution is calculated when using MRF?
2. When using rotating mesh, which form of velocity is used when solving the velocity equations, absolute or relative?

How to modify it:

1. If creating an own copy of `finiteVolume`, but only including MRF, using the same file structure as the original library, how would the directory tree look?

Appendix A

Complete codes

A.1 setMRFFaces in MRFZone.C

The function setMRFFaces in MRFZone.C

```
void Foam::MRFZone::setMRFFaces()
{
    const polyBoundaryMesh& patches = mesh_.boundaryMesh();

    // Type per face:
    //  0:not in zone
    //  1:moving with frame
    //  2:other
    labelList faceType(mesh_.nFaces(), Zero);

    // Determine faces in cell zone
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // (without constructing cells)

    const labelList& own = mesh_.faceOwner();
    const labelList& nei = mesh_.faceNeighbour();

    // Cells in zone
    boolList zoneCell(mesh_.nCells(), false);

    if (cellZoneID_ != -1)
    {
        const labelList& cellLabels = mesh_.cellZones()[cellZoneID_];
        forAll(cellLabels, i)
        {
            zoneCell[cellLabels[i]] = true;
        }
    }


    label nZoneFaces = 0;

    for (label facei = 0; facei < mesh_.nInternalFaces(); facei++)
    {
        if (zoneCell[own[facei]] || zoneCell[nei[facei]])
        {
            faceType[facei] = 1;
            nZoneFaces++;
        }
    }


    labelHashSet excludedPatches(excludedPatchLabels_);

```

```
    forAll(patches, patchi)
    {
        const polyPatch& pp = patches[patchi];

        if (pp.coupled() || excludedPatches.found(patchi))
        {
            forAll(pp, i)
            {
                label facei = pp.start()+i;

                if (zoneCell[own[facei]])
                {
                    faceType[facei] = 2;
                    nZoneFaces++;
                }
            }
        }
        else if (!isA<emptyPolyPatch>(pp))
        {
            forAll(pp, i)
            {
                label facei = pp.start()+i;

                if (zoneCell[own[facei]])
                {
                    faceType[facei] = 1;
                    nZoneFaces++;
                }
            }
        }
    }

    // Synchronize the faceType across processor patches
    syncTools::syncFaceList(mesh_, faceType, maxEqOp<label>());

    // Now we have for faceType:
    //  0   : face not in cellZone
    //  1   : internal face or normal patch face
    //  2   : coupled patch face or excluded patch face

    // Sort into lists per patch.

    internalFaces_.setSize(mesh_.nFaces());
    label nInternal = 0;

    for (label facei = 0; facei < mesh_.nInternalFaces(); facei++)
    {
        if (faceType[facei] == 1)
        {
            internalFaces_[nInternal++] = facei;
        }
    }
    internalFaces_.setSize(nInternal);

    labelList nIncludedFaces(patches.size(), Zero);
    labelList nExcludedFaces(patches.size(), Zero);

    forAll(patches, patchi)
    {
        const polyPatch& pp = patches[patchi];

        forAll(pp, patchFacei)
        {
            label facei = pp.start() + patchFacei;

            if (faceType[facei] == 1)
            {
                nIncludedFaces[patchi]++;
            }
            else if (faceType[facei] == 2)
            {
                nExcludedFaces[patchi]++;
            }
        }
    }

    includedFaces_.setSize(patches.size());
    excludedFaces_.setSize(patches.size());
    forAll(nIncludedFaces, patchi)
    {
        includedFaces_[patchi].setSize(nIncludedFaces[patchi]);
        excludedFaces_[patchi].setSize(nExcludedFaces[patchi]);
    }
    nIncludedFaces = 0;
    nExcludedFaces = 0;

    forAll(patches, patchi)
    {
        const polyPatch& pp = patches[patchi];

        forAll(pp, patchFacei)
        {
            label facei = pp.start() + patchFacei;

            if (faceType[facei] == 1)
            {
                includedFaces_[patchi][nIncludedFaces[patchi]++] = patchFacei;
            }
            else if (faceType[facei] == 2)
            {
                excludedFaces_[patchi][nExcludedFaces[patchi]++] = patchFacei;
            }
        }
    }


    if (debug)
    {
        faceSet internalFaces(mesh_, "internalFaces", internalFaces_);
        Pout<< "Writing " << internalFaces.size()
            << " internal faces in MRF zone to faceSet "
            << internalFaces.name() << endl;
        internalFaces.write();

        faceSet MRFFaces(mesh_, "includedFaces", 100);
        forAll(includedFaces_, patchi)
        {
            forAll(includedFaces_[patchi], i)
            {
                label patchFacei = includedFaces_[patchi][i];
                MRFFaces.insert(patches[patchi].start()+patchFacei);
            }
        }
        Pout<< "Writing " << MRFFaces.size()
            << " patch faces in MRF zone to faceSet "
            << MRFFaces.name() << endl;
        MRFFaces.write();

        faceSet excludedFaces(mesh_, "excludedFaces", 100);
        forAll(excludedFaces_, patchi)
        {
            forAll(excludedFaces_[patchi], i)
            {
                label patchFacei = excludedFaces_[patchi][i];
                excludedFaces.insert(patches[patchi].start()+patchFacei);
            }
        }
        Pout<< "Writing " << excludedFaces.size()
            << " faces in MRF zone with special handling to faceSet "
            << excludedFaces.name() << endl;
        excludedFaces.write();
    }
}
```

Appendix B

Test case dictionaries

B.1 blockMeshDict

blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

scale   1;

vertices
(
    (-2 -1 -0.199)
    (3 -1 -0.199)
    (3  1 -0.199)
    (-2  1 -0.199)
    (-2 -1 2)
    (3 -1 2)
    (3  1 2)
    (-2  1 2)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (30 12 14) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    frontAndBack
    {
        type patch;
        faces
        (
            (3 7 6 2)
            (1 5 4 0)
        );
    }
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    lowerWall
    {
        type wall;
        faces
        (
            (0 3 2 1)
        );
    }
    upperWall
    {
        type patch;
        faces
        (
            (4 5 6 7)
        );
    }
);

// ************************************************************************* //
```

B.2 surfaceFeatureExtractDict

surfaceFeatureExtractDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      surfaceFeatureExtractDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

tyre_grooves_flat.stl
{
    extractionMethod      extractFromSurface;
    includedAngle         150;

    subsetFeatures
    {
        nonManifoldEdges  no;
        openEdges         yes;
    }

    writeObj              yes;
}


// ************************************************************************* //
```

B.3 snappyHexMeshDict

snappyHexMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      snappyHexMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       true;

geometry
{
    tyre_grooves_flat.stl
    {
        type triSurfaceMesh;
        name tyre;

        regions
        {
            tyre
            {
                name tyre;
            }
        }
    }

    refinementBox
    {
        type   box;
        min    (-0.5 -0.3 -0.2);
        max    (1.0  0.3  0.5);
    }
}

castellatedMeshControls
```

```
{
    maxLocalCells 100000;
    maxGlobalCells 2000000;
    minRefinementCells 1000;
    nCellsBetweenLevels 4;

    // Explicit feature edge refinement
    features
    (
        {
            file "tyre_grooves_flat.eMesh";
            level 6;
        }
    );

    // Surface based refinement
    refinementSurfaces
    {
        tyre
        {
            level (5 5);
        }
    }

    resolveFeatureAngle 30;

    // Region-wise refinement
    refinementRegions
    {
        refinementBox
        {
            mode inside;
            levels ((1e15 4));
        }
    }

    // Mesh selection
    locationInMesh (1.001 0.001 0.501);

    allowFreeStandingZoneFaces true;
}

snapControls
{
    nSmoothPatch 3;
    tolerance 2.0;
    nSolveIter 30;
    nRelaxIter 5;

    // Feature snapping
    nFeatureSnapIter 10;
    implicitFeatureSnap false;
    explicitFeatureSnap true;
    multiRegionFeatureSnap false;
}

addLayersControls
{
    relativeSizes true;

    layers
    {
        "(tyre|tyre_grooves).*"
        {
            nSurfaceLayers 1;
        }
    }

    expansionRatio 1.0;
    finalLayerThickness 1.0;
    minThickness 0.1;
    nGrow 0;

    // Advanced settings
    featureAngle 30;
    nRelaxIter 3;
    nSmoothSurfaceNormals 1;
    nSmoothNormals 3;
    nSmoothThickness 10;
    maxFaceThicknessRatio 0.5;
    maxThicknessToMedialRatio 0.3;
    minMedialAxisAngle 90;
    nBufferCellsNoExtrude 0;
    nLayerIter 50;
    nRelaxedIter 20;
}

meshQualityControls
{
    maxNonOrtho 65;
    maxBoundarySkewness 20;
    maxInternalSkewness 4;
    maxConcave 80;
    minVol 1e-20;
    minTetQuality 1e-30;
    minArea -1;
    minTwist 0.02;
    minDeterminant 0.001;
    minFaceWeight 0.02;
    minVolRatio 0.01;
    minTriangleTwist -1;
    nSmoothScale 4;
    errorReduction 0.75;
    relaxed
    {
        // Maximum non-orthogonality allowed. Set to 180 to disable.
        maxNonOrtho 65;
    }
}

mergeTolerance 1e-6;

// ************************************************************************* //
```

Index