CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

A tutorial on modification of the turboFvMesh class for flow-driven rotation

---

Developed for foam-extend 3.1

Author:
Erik Krane

Peer reviewed by:
Jonatan Margalit
Håkan Nilsson

January 22, 2015

Chapter 1

Theory and current implementations

1.1 Introduction

In foam-extend 3.1 there is a class that handles mesh movement in the form of pure rotation (no translation or deformation); this class is called `turboFvMesh`. Today the `turboFvMesh` class handles constant rotational speeds of one or several rotors about an axis of rotation. There also exists several classes that allows to solve for the rigid body motion of several bodies. In the present work these two functionalities will be combined to create a dynamic mesh class that will allow a variable rotational speed based on the fluid forces and a requested shaft power.

1.2 Theory

The theory and mathematics of this work is not very advanced, but to make all parts interact can be a challenge. The different parts of this project include force calculations, rigid body dynamics and mesh motion.

1.2.1 Fluid forces

To calculate the rotational acceleration of an object due to fluid forces, the moment the fluid is exerting on that object needs to be calculated. This is done using the function object `forces`, which calculates the forces and moments on a given boundary. The force density is defined as

$$\mathbf{f}_D = -p + \tau_w, \tag{1.1}$$

where p is the pressure and τ_w is the wall shear stress. The normal force is then given by

$$\mathbf{f}_N = \frac{\mathbf{S}_{fb}}{|\mathbf{S}_{fb}|} \left(\mathbf{S}_{fb} \cdot \mathbf{f}_D \right), \tag{1.2}$$

where $\mathbf{S}_{fb}$ is the surface normal vector of each face. The forces for every face are summed up to get the total pressure force. The moment arm, $\mathbf{r}$ and the face element force form a cross product that is then summed to get the moment exerted by the normal forces.

$$\mathbf{F}_{pressure} = \sum_{faces} \mathbf{f}_N \tag{1.3}$$

$$\mathbf{M}_{pressure} = \sum_{faces} \left(\mathbf{r} \times \mathbf{f}_N \right) \tag{1.4}$$

The normal force is subtracted from the total force to obtain the tangential force on each face.

$$\mathbf{f}_T = |\mathbf{S}_{fb}|\mathbf{f}_D - \mathbf{f}_N \tag{1.5}$$

Analogous to the operations in equations 1.3 and 1.4 the total shear force and moment are obtained.

$$\mathbf{F}_{shear} = \sum_{faces} \mathbf{f}_T \tag{1.6}$$

$$\mathbf{M}_{shear} = \sum_{faces} (\mathbf{r} \times \mathbf{f}_T) \tag{1.7}$$

The moment about a specified axis is required in the present work. If the unit vector $\mathbf{e} = [e_x, e_y, e_z]$ is the axis of rotation, the moment that accelerates the rotation is given by the scalar product

$$M_{flow} = (\mathbf{M}_{pressure} + \mathbf{M}_{shear}) \cdot \mathbf{e}. \tag{1.8}$$

1.2.2 Rigid body mechanics

The angular acceleration, α, of a rigid body will depend on the sum of the moments affecting the body and the moment of inertia of the body, $\mathbf{I}$, as

$$\sum \mathbf{M} = \alpha \mathbf{I}. \tag{1.9}$$

In the present work the rotational speed will be limited by the power output, P, this will create a moment

$$M_{power} = \frac{P}{\omega}, \tag{1.10}$$

that balances the moment from the fluid forces and gives the total acceleration as

$$\alpha = \frac{M_{flow} - M_{power}}{I}, \tag{1.11}$$

the rotational speed can then be calculated as

$$\omega = \int_{t_{old}}^{t} \alpha dt = \omega_{old} + \alpha \Delta t. \tag{1.12}$$

The equations 1.10-1.12 are scalar equations as rotation about the z-axis is the only rotation supported in the present work.

1.3 Existing foam-extend implementations

1.3.1 The turboFvMesh dynamic mesh class

The `turboFvMesh` class, defined in `$FOAM_SRC/dynamicMesh/dynamicFvMesh/turboFvMesh`, is used for rotating mesh displacement and will not translate or deform the mesh. It uses a prescribed, constant rotational speed to rotate one or several mesh regions about an axis. It has three member functions, `update()`, `movingPoints()` and `calcMovingPoints()`. The `update()` function exists in all dynamic mesh classes and sends the updated mesh to the solver. The `movingPoints()` function checks if the `movingPointsPtr_` pointer is false and in that case calls `calcMovingPoints()`. The `calcMovingPoints()` does all the calculations. It goes through `dynamicMeshDict`, looks up the rotating zones and calculates a mesh motion based on the rpm value. The `dynamicMeshDict` dictionary is used for most dynamic mesh classes and for `turboFvMesh` it looks like:

```
dynamicFvMesh       turboFvMesh;

turboFvMeshCoeffs
{
    coordinateSystem
    {
        type            cylindrical;
        origin          (0 0 0);
        axis            (0 0 1);
        direction       (1 0 0);
    }

    rpm
    {
        Rotor1 60;
        Rotor2 -30;
    }

    slider
    {
        Rotor1_faceZone 60;
        Rotor2_faceZone -30;
    }
}
```

Figure 1.1: Example of dynamicFvMesh for turboFvMesh

The coordinate system is defined as well as the rotational speeds for the different rotating zones (`Rotor1` and `Rotor2` in this case). The sub-dictionary `slider` defines the rotational speed of the sliding interfaces connected to the mesh zones.

1.3.2 The sixDOF libraries of foam-extend

In foam-extend there are several libraries and classes that are used to calculate and apply rigid body motion to bodies or meshes. In table 1.1 the classes are presented. All of the classes solve the rigid body motion of one or several bodies. The main difference between the sixDOF classes based on the ODE library and the function objects is the input and output. In `sixDOFsolver` no mesh is used and all the computational data is stored in a dictionary that is written every time step. The solver `sixDOFsolver` uses `sixDOFbodies` which in turn uses `sixDoFqODE`. This is the class that is actually solving the rigid body equations and updates the dictionary. The function object `sixDoFRigidbodyDisplacement` works in a different way, calculating displacement of boundary patches and will morph the mesh. If this is not desirable, as in the case of the present work, the class can still be of use. In this case the code snippet calculating the forces and moments was used, as these calculations are done on the boundary patches.

Table 1.1: Description of implemented sixDOF-classes

Class or application	**Description**
`sixDOFsolver`	Solver that uses the ODE libraries `sixDOFbodies` and `sixDOFqODE` to solve equations of motion for a dynamic problem. Uses no mesh, but one dictionary for each body that is solved for.
`sixDOFbodies`	Is used in the `sixDOFsolver` to calculate the rigid body motion of one or several bodies. Uses `sixDOFqODE` to find the translation and rotation of the bodies.
`sixDoFqODE`	An ODE class that calculates the motion of one or several bodies using quaternions.
`sixDoFRigidbodyDisplacement`	Solves equations of motion for boundary patches (applied as boundary condition) by instantiating the `forces` function object to find the forces and moments on the boundary patches. `sixDoFRigidBodyMotion` is used to solve the equations of motion. The mesh is updated, *and* deformed.
`sixDoFRigidBodyMotion`	A function object that is used in `sixDoFRigidbodyDisplacement` to calculate rigid body motion.

Chapter 2

Class modification

2.1 Modification of turboFvMesh

To be able to dynamically change the rotational speed of a rotating mesh region, `turboFvMesh` needs to be modified in several ways. Forces on the rotating regions have to be calculated, an updated rotation has to be calculated from these forces and passed to the mesh movement function. The original `turboFvMesh` only needs to calculate rotation once and keep it constant, as the class is written for this purpose and will not update the rotational speed continuously.

2.1.1 Copy and rename the original class

Start by activating the foam-extend 3.1 environment, then copy the original class to your own directory and rename it to the name of your choice. Here, the name `turboVarRpmFvMesh` is chosen. Also, remove files that are not needed.

```
cd $WM_PROJECT_DIR
cp -r --parents src/dynamicMesh/dynamicFvMesh/turboFvMesh $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/dynamicMesh/dynamicFvMesh
mv turboFvMesh turboVarRpmFvMesh
cd turboVarRpmFvMesh
mv turboFvMesh.C turboVarRpmFvMesh.C
mv turboFvMesh.H turboVarRpmFvMesh.H
rm dynamicMeshDict *.dep
```

The names inside the files have to be changed as well.

```
sed -i s/turboFvMesh/turboVarRpmFvMesh/g turboVarRpmFvMesh.*
```

Now, the a Make-directory, `Make/files` and `Make/options` need to be created. These should be located in the `dynamicFvMesh` directory to follow the structure of the original class.

```
cd ..
mkdir Make
touch Make/files
touch Make/options
```

`Make/files` should only include the .C-file that was just created and point to a new library.

```
turboVarRpmFvMesh/turboVarRpmFvMesh.C
LIB = $(FOAM_USER_LIBBIN)/libmyDynamicFvMesh
```

In `Make/options` the libraries needed for the compilation are added.

```
EXE_INC = \
    -I$(LIB_SRC)/dynamicMesh/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/postProcessing/functionObjects/forces/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -ldynamicMesh \
    -ldynamicFvMesh \
    -lfiniteVolume \
    -lforces \
    -lmeshTools
```

Now, the new class should be ready to compile with the same functionality as the original class.

```
wclean
wmake libso
```

If it compiles with no errors the next step will be to go back into `turboVarRpmFvMesh` and modify the contents to include the variable rotational speed.

```
cd turboVarRpmFvMesh
```

2.1.2 Modification of turboVarRpmFvMesh.C

The source file `turboVarRpmFvMesh.C` is the first one to be changed. Two new header files need to be added. They are needed to calculate the forces and to write an output file.

```
#include "forces.H"
#include <iostream>
```

The code should be changed so that the rotational speed is updated every time step. In the private member function `calcMovingPoints()` belonging to `turboVarRpmFvMesh.C` the code that prohibits the updating is removed. The const is removed to make the function variable. In the following instructions the text in red is removed.

```
void Foam::turboVarRpmFvMesh::calcMovingPoints()const
```

This piece of code tells the function to abort if `movingPointsPtr_` have been calculated before, so in order for the rpm to be repeatedly updated it has to be removed.

```
56 if (movingPointsPtr_)
57 {
58     FatalErrorIn("void turboFvMesh::calcMovingMasks() const")
59     << "point mask already calculated"
60     << abort(FatalError);
61 }
```

Changes made to the code are marked by a green color. The rotational speed from the last time step, `rpmOld`, needs to be declared and read from the dictionary. It is used later to calculate the new rpm according to equation 1.12. This is done in the file `variableRpm.H` that is created and

explained later. The `rpmOutput.H` file is used to write a data file that can be monitored using the plotting software of your choice. It will be explained later as well. In the next line the calculated rpm value is added to a dictionary to be used in the next time step as `rpmOld`.

```
56      scalar rpm;
57      scalar rpmOld;
58
59      forAll (cellZoneNames,cellZoneI)
60      {
61          const labelList& cellAddr =
62              cellZones()[cellZones().findZoneID(cellZoneNames[
    cellZoneI])];
63
64          if (dict_.subDict("rpm").found(cellZoneNames[cellZoneI]))
65          {
66              rpmOld = readScalar
67              (
68                  dict_.subDict("rpm").lookup(cellZoneNames[cellZoneI
    ])
69              );
70
71              #include "variableRpm.H"
72              #include "rpmOutput.H"
73
74              dict_.subDict("rpm").add(cellZoneNames[cellZoneI], rpm,
    true);
75
76              Info<< "Moving Cell Zone Name: " << cellZoneNames[
    cellZoneI]
77                  << " rpm: " << rpm << endl;
```

Change the slider rpm to that of the associated mesh zone. The changed line looks up the name of the rotating mesh zone associated with the sliding interface and applies that rpm value.

```
131 if (dict_.subDict("slider").found(faceZoneNames[faceZoneI]))
132         {
133             rpm = readScalar(dict_.lookup(
134                     dict_.subDict("slider").lookup(faceZoneNames[
    faceZoneI])));
135
136             Info<< "Moving Face Zone Name: " << faceZoneNames[
    faceZoneI]
137                 << " rpm: " << rpm << endl;
138
139             faceZoneID zoneID(faceZoneNames[faceZoneI], faceZones())
    ;
```

The member function `movingPoints()` also need some modification. There is an if-statement that calls the function `calcMovingPoints()` if `movingPointsPtr_` is false. As `calcMovingPoints()` should be called every time step the if-statement is removed. The const designation also have to be removed to enable updating of the function.

```
216 const Foam::vectorField& Foam::turboVarRpmFvMesh::movingPoints()
    const
217 {
218     if (!movingPointsPtr_)
```

```
219     {
220         calcMovingPoints();
221     }
222     return *movingPointsPtr_;
223 }
```

The already existing IOdictionary `dict_` is used to read and write the `rpm` value of the current time step. This is the dictionary where `rpmOld` is read from. The only change needed here is to change the last `IOobject` to `AUTO_WRITE` so that the rpm value can be written.

```
164 dict_
165     (
166         IOdictionary
167         (
168             IOobject
169             (
170                 "dynamicMeshDict",
171                 time().constant(),
172                 *this,
173                 IOobject::MUST_READ,
174                 IOobject::AUTO_WRITE
175             )
176         ).subDict(typeName + "Coeffs")
177     ),
```

2.1.3 The variableRpm.H file

Now, the file `variableRpm.H` is created, it is used to calculate the new rpm from defined values in the `dynamicMeshDict` dictionary and the forces on the rotating patches.

```
vim variableRpm.H
```

Variables are declared and defined, some variables are loaded from the dictionary.

```
const scalar Pi(3.14159265359);
const vector CofR = dict_.subDict("coordinateSystem").lookup("origin");
const scalar rhoInf = readScalar( dict_.subDict("variableRpm").lookup("rhoInf") );
const wordList patches = dict_.subDict("forcePatches").lookup(cellZoneNames[cellZoneI]);
const scalar Iz = readScalar(dict_.subDict("variableRpm").lookup("Iz"));
const scalar cutoffRpm = readScalar(dict_.subDict("variableRpm").lookup("cutoffRpm"));

scalar Pnom = readScalar( dict_.subDict("variableRpm").lookup("Pnom") );
scalar rpmNom = readScalar( dict_.subDict("variableRpm").lookup("rpmNom") );
scalar P;
scalar alpha;
scalar Mflow;
scalar Mpow;
```

A short explanation of the variables used can be found below.

Pi - The geometrical constant π.

CofR - The centre of rotation of the rotor.

rhoInf - The density of the free stream flow.

patches - The patches on which the fluid forces are calculated

Iz - Moment of inertia about the z-axis

cutoffRpm - A limiting rpm under which the power is set to zero.

Pnom - Nominal power extracted from the rotor.

rpmNom - Nominal rotational speed of the rotor.

P - Power used to calculate the extracted moment.

Mflow - The moment the fluid forces give rise to.

Mpow - The extracted (shaft) moment.

The `forces` function object is used to calculate the moment on the patches as in equation 1.8. This is done by integration of the pressure and viscous forces over the patch area to get a force which is then cross multiplied with the position vector, **r**, from the centre of rotation to get the moment.

Below, the code calculating the flow moment is presented, note that we are still in the `variableRpm.H` file. In the variable `fm` the forces and moments split in a pressure and a viscous part. The two parts of the moments are added to form `Mflow`. Before the calculations are done `forces` need some information. This is added through the definition of a dictionary called `forcesDict`. The parameters in `forcesDict` are all defined in the sub-dictionary `variableRpm` in `dynamicMeshDict` except `forcePatches` which has got its own sub-dictionary.

```
dictionary forcesDict;
forcesDict.add("patches", patches);
forcesDict.add("rhoName", "rhoInf");
forcesDict.add("rhoInf", rhoInf);
forcesDict.add("CofR", CofR);

forces F("forces", *this, forcesDict);
forces::forcesMoments fm = F.calcForcesMoment();
Mflow = fm.second().first().z()+fm.second().second().z();
```

The moment about the axis of rotation and the prescribed power output are used to calculate the angular acceleration, which is then multiplied with Δt and added to the rotational speed (eq. 1.12) of the last time step to get the new rpm. The cutoff speed can be used if there is a need to limit the shaft power below a specific rotational speed.

```
// Apply moment if the rotational speed is higher than cutoff rpm
if (abs(rpmOld) > abs(cutoffRpm))
{
    P = Pnom;
}
// P=0 for low rpm:s
else
{
    P = 0;
}

Mpow = Pnom/(rpmNom*Pi/30);

// Calculate angular acceleration and rpm
// (with accleration limiter for numerical stability)
alpha = ( Mflow-Mpow )/Iz;
```

```
if (abs(alpha) > 5)
{
    if (pos(alpha))
    {
        alpha = 5;
    }
    else if (neg(alpha))
    {
        alpha = -5;
    }
}

rpm = rpmOld + ( (30/Pi)*alpha )*time().deltaT().value();
```

2.1.4 Modification of turboVarRpmFvMesh.H

In the header file `turboVarRpmFvMesh.H` only two lines have to be changed, the *const* designation is removed from the declaration of the member functions `calcMovingPoints()` and `movingPoints()`.

```
99  //- Calculate moving Points
100 void calcMovingPoints() const;
101
102 //- Return moving points
103 const vectorField& movingPoints() const;
```

2.1.5 The rpmOutput.H file

The file `rpmOutput.H` is created, it is used to write the file "rpmData.dat" to the case directory. The file "rpmData.dat" contains the rpm value for every time step and can be used to plot the rpm while the simulation is running. The `std::` namespace is used to enable the use of the append mode in a simple way.

```
if (Pstream::master())
{
    fileName rpmDataDir;
    if (Pstream::parRun())
    {
        rpmDataDir = time().path() + "/../";
    }
    else
    {
        rpmDataDir = time().path();
    }
    word tracingFileName = "rpmData.dat";
    fileName rpmDataFile = rpmDataDir/tracingFileName;

    std::ofstream os
    (
        rpmDataFile.c_str(),
        ios_base::app
    );
    os << time().value() << tab
       << rpm   <<"\n";
}
```

Now `turboVarRpmFvMesh` is ready for compilation. Navigate to the directory where the `Make` folder is located and run the commands:

```
wclean
wmake libso
```

If it runs without errors the class is now ready to use.

2.2 Running the axialTurbine tutorial

The new class will be tested using the `axialTurbine` tutorial case. The case features one passage of a turbine with stator and rotor. It uses the solver `pimpleDyMFoam` with a General Grid Interface to handle the coupling between the rotating and the stationary mesh regions. The geometry of the turbine can be seen in figure 2.1 and the inlet boundary condition is U = 1 m/s in negative z-direction and pressure is zero gradient. At the outlet the velocity boundary condition is zero gradient and the pressure is zero. At the interfaces between stator-rotor and rotor-draft tube the `overlapGGI` boundary condition is used which interpolates the values between the mesh zones and the `cyclicGgi` is used at the cyclic sides.

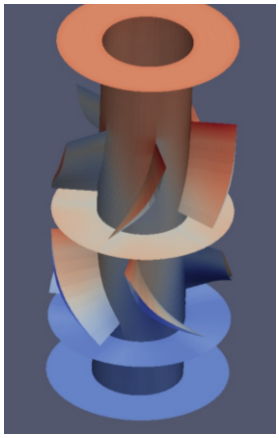

Figure 2.1: The axialTurbine tutorial case copied rotationally. Only one blade row is solved for in this case. Here, the rotor and the stator are coloured by pressure

2.2.1 Copy and modify the case

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/pimpleDyMFoam/axialTurbine .
cd axialTurbine
```

As the newly created `turboVarRpmFvMesh` class is to be used the `dynamicMeshDict` should be modified.

```
vim constant/dynamicMeshDict
```

The values are set to be suitable for the `axialTurbine` tutorial case and would be different for other applications.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     3.1                                |
|   \\  /    A nd           | Web:         http://www.extend-project.de       |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dynamicFvMesh       turboVarRpmFvMesh;

turboVarRpmFvMeshCoeffs
{
    coordinateSystem
    {
        type            cylindrical;
        origin          (0 0 0);
        axis            (0 0 1);
        direction       (1 0 0);
    }
    rpm
    {
       rotor  -50;    // initial rpm
    }
    slider
    {
        RUINLET   "rotor";
        RUOUTLET  "rotor";
        RUCYCLIC1 "rotor";
        RUCYCLIC2 "rotor";
    }
    forcePatches
    {
        rotor    ( RUHUB RUBLADE );
    }
    variableRpm
    {
        rhoInf    1000;    // Free stream density
        Iz        0.1;     // Moment of inertia
        cutoffRpm 90;      // rpm under which P=0
        Pnom      3;    // Nominal power and rpm used to calculate nominal moment
        rpmNom    -95;
    }
}
```

One thing to note about the entries `Pnom` and `rpmNom` is the notation "nominal". Because of the need to keep the extracted moment constant rather than the power this notation is used. If the power is kept constant and the moment from the flow forces becomes higher than that of the extracted moment the rotational speed will increase. This will lead to a decrease in extracted moment, which further increases the rotational speed until M_{power} becomes very small; hence the use of a constant $M_{power} = P_{nom}/\omega_{nom}$.

To be able to use the new class the library it is included in need to be defined in `system/controlDict`. This is done by adding the following line at the end of `controlDict`.

```
libs ("libmyDynamicFvMesh.so");
```

To monitor the rotational speed while the the solver is running a simple script can be used. Create the file `gnuRead` including the following lines.

```
plot "rpmData.dat" using 1:2 with lines
pause 1
reread
```

This is all modifications that are needed to run the case. Run the case by executing the `Allrun` script and putting the output in a log file.

```
./Allrun >& log &
```

Monitor the rotational speed with gnuplot using the command:

```
gnuplot gnuRead
```

2.2.2 Results

In the figures 2.2-2.4 two periods of rotation of the rotational speed, angular acceleration and moments are plotted after a simulation time of about 30 s. The frequency analysis of the angular acceleration is shown in figure 2.5, where there are two main frequencies present. The lowest one f_1=1.58 Hz is close to that of the mean rotational speed of 90 rpm, $f = \frac{\omega}{2\pi}$=1.5 Hz. The higher f_2=7.76 Hz is close to the blade passing frequency $f = 5 \times 1.5$=7.5 Hz. This deviation of 5.3% and 3.5%, respectively, from the theoretical indicates that there are some real physics represented by the current modifications.

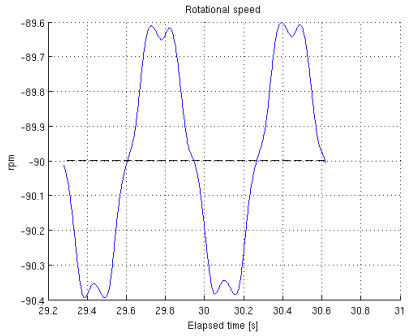

Figure 2.2: Rotational speed for two rotations where the dashed black line is the mean.

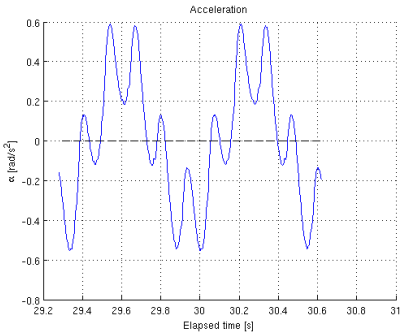

Figure 2.3: Angular acceleration for two rotations where the dashed black line is the mean.

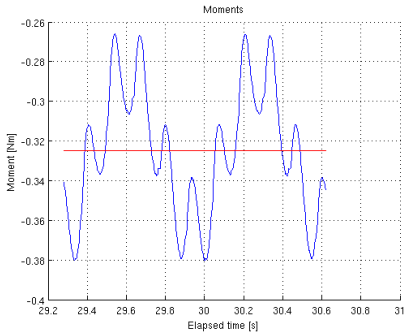

Figure 2.4: M_{flow} (blue) and M_{power} (red) for two rotations

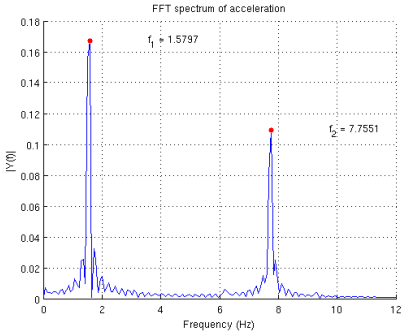

Figure 2.5: FFT of the angular acceleration for 15 rotations