CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Simplified flow around a propeller

Developed for OpenFOAM-2.4.x

Author:
Gonzalo Montero

Peer reviewed by:
Bjarke Eltard-Larsen
Håkan Nilsson

February 4, 2016

Chapter 1

1.1 Learning outcomes

The reader will learn:

- to implement and use the solver *propellerSimpleFoam*.
- to update Erik Svenning's (course of 2010) work from OpenFOAM 1.5-dev to 2.4x
- general knowledge about actuator disks and propellers
- some theory regarding propeller performance prediction
- what *Xfoil* can be used for
- to call *Xfoil* from the code
- to use m4 to parametrize *blockMeshDict*

1.2 Introduction and motivation

This work is built on top of Erik Svenning's work [Svenning, 2010]. Erik implemented an actuator disk to model the flow around a propeller. The flow was calculated among other things from a given thrust, torque and prescribed force distribution around the blade (an approximation to the optimum force distribution defined by Goldstein). The idea behind this new implementation is to be able to analyze a more realistic propeller by taking into account its geometry, operating conditions and so on. So first lets start by taking a look at Erik's implementation. Erik's original work file can be downloaded here:
`www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/erikSvenning/erikSvenningFiles.tgz`
and the report:
`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/erikSvenning/erikSvenningReport.pdf`
and the files corresponding to the present report:
`www.tfd.chalmers.se/~hani/kurser/OS_CFD/GonzaloMonteroVillar/gonzaloMonteroFiles.tgz`

1.3 actuatorDiskExplicitForceSimpleFoam by Erik Svenning updated to OpenFOAM 2.4.x

The *actuatorDiskExplicitForceSimpleFoam* solver was originally developed for *OpenFOAM 1.5-dev*, thus, to make it work in *OpenFOAM 2.4.x* version, some small adjustments are required. First download and copy the files into your desktop, and enter that folder.

```
wget www.tfd.chalmers.se/~hani/kurser/OS_CFD/GonzaloMonteroVillar/gonzaloMonteroFiles.tgz
tar -zxvf gonzaloMonteroFiles.tgz
rm gonzaloMonteroFiles.tgz
mv gonzaloMonteroFiles/ ~/Desktop
cd ~/Desktop/gonzaloMonteroFiles/
```

Then copy the *readSIMPLEControls.H* file from the *Additional files* folder to *erikSvenningOriginalFiles/actuatorDiskExplicitForce* and go into that folder

```
cp Additional\ files/readSIMPLEControls.H erikSvenningOriginalFiles/actuatorDiskExplicitForce/
cd erikSvenningOriginalFiles/actuatorDiskExplicitForce/
```

Execute (be careful if copy pasting because single quotation is interpreted as a new line by the terminal)

```
sed -i 's+#include "incompressible/RASModel/RASModel.H"+#include
    "incompressible/RAS/RASModel/RASModel.H"+' actuatorDiskExplicitForceSimpleFoam.C
```

to modify the location where the file *RASModel.H* is located on the file *actuatorDiskExplicitForceSimpleFoam.C*. Then change the content in the *Make/options* file to:

```
EXE_INC = \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/RAS/RASModel \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/RAS \
    -I$(LIB_SRC)/finiteVolume/cfdTools/general/include

EXE_LIBS = \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleTransportModels \
    -lfiniteVolume \
    -lmeshTools \
    -lfvOptions \
    -lsampling
```

Finally type:

```
OF24x
wmake
```

to compile *actuatorDiskExplicitForceSimpleFoam* solver. Now lets do the modifications necessary to run the case.

```
cd ../caviyActuatorDisk/system
```

In *fvSchemes* in order to update it execute:

```
sed -i 's+div((nuEff\*dev(grad(U).T()))) Gauss linear;+
    div((nuEff\*dev(T(grad(U))))) Gauss linear;+' fvSchemes
```

and also chanfe the divergence schemes to *bounded Gauss* in order to avoid a warning by executing (be careful with the spacing):

```
sed -i 's+div(phi,U)      Gauss upwind;+div(phi,U)      bounded Gauss upwind;+' fvSchemes
sed -i 's+div(phi,k)      Gauss upwind;+div(phi,k)      bounded Gauss upwind;+' fvSchemes
sed -i 's+div(phi,epsilon) Gauss upwind;+div(phi,epsilon) bounded Gauss upwind;+' fvSchemes
```

and run the case:

```
cd ..
blockMesh
actuatorDiskExplicitForceSimpleFoam
```

Finally the result can be visualized using *paraFoam* by executing:

```
paraFoam
```

NOTE: not much is going to appear in the results since it has been run for only a few iterations

1.4 Theory background and tools used

This section presents the equations used to compute the thrust and torque produced at each radial station of the blade. The propeller blade will be discretised into a finite number of elements that will be treated individually to calculate their contribution to the thrust and torque produced. To calculate the total thrust and torque all of them are summed up.

1.4.1 Nomenclature

r	*radial coordinate*
R	*tip radius*
T	*thrust*
Q	*torque*
$\beta(r)$	*local blade pitch angle*
$\phi(r)$	*local flow angle*
$\alpha(r)$	*local angle of attack*
$\Gamma(r)$	*local blade circulation*
B	*number of blades*
V	*freestream velocity*
Ω	*rotational speed*
$c(r)$	*local blade chord*
$W(r)$	*local relative velocity*
$c_l(r)$	*local blade lift coefficient*
$c_d(r)$	*local blade drag coefficient*
λ	*advance ratio* $(= V/\omega R)$
$\lambda_w(r)$	*local wake advance ratio*$(= rW_a/(RW_t))$
ρ	*fluid density*
μ	*fluid viscosity*
a	*fluid speed of sound*
$()_a,\ ()_t$	*axial and tangential components*

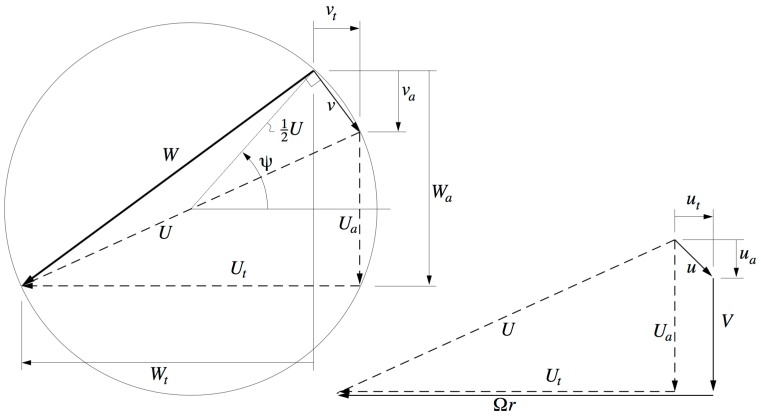

Figure 1.1: Velocity parametrization by ψ

1.4.2 Obtaining thrust and torque

Firstly, all the velocities are going to be expressed with respect to the angle ψ. This can be seen in Fig. 1.1. Newton method will be used to solve iteratively starting from a guessed value of ψ. The system of equations for which the iterative method is solved reads:

$$U_a = V + u_a \tag{1.1}$$
$$U_t = \Omega r - u_t \tag{1.2}$$
$$U = \sqrt{U_a^2 + U_t^2} \tag{1.3}$$
$$W_a(\psi) = 0.5U_a + 0.5Usin\psi \tag{1.4}$$
$$W_t(\psi) = 0.5U_t + 0.5Ucos\psi \tag{1.5}$$
$$v_a(\psi) = W_a - U_a \tag{1.6}$$
$$v_t(\psi) = U_t - W_t \tag{1.7}$$
$$\alpha(\psi) = \beta - arctan(W_a/W_t) \tag{1.8}$$
$$W(\psi) = \sqrt{W_a^2 + W_t^2} \tag{1.9}$$
$$\lambda_w(\psi) = \frac{rW_a}{RW_t} \tag{1.10}$$
$$f(\psi) = 0.5B\left(1 - \frac{r}{R}\right)\frac{1}{\lambda_w} \tag{1.11}$$
$$F(\psi) = \frac{2}{\pi}arccos(e^{-f}) \tag{1.12}$$
$$\Gamma(\psi) = v_t\frac{4\pi r}{B}F\sqrt{1 + \left(\frac{4\lambda_w R}{\pi Br}\right)^2} \tag{1.13}$$

We can also get the circulation as:

$$\Gamma = \frac{1}{2}Wcc_l \tag{1.14}$$

Finally with the two last relations for the circulation (Eqs. 1.13 and 1.14) we can establish a residual used for the aforementioned Newton iterations as:

$$R(\psi) = v_t\frac{4\pi r}{B}F\sqrt{1 + \left(\frac{4\lambda_w R}{\pi Br}\right)^2} - \frac{1}{2}Wcc_l \tag{1.15}$$

The Newton update for ψ then reads:

$$\delta\psi = -\frac{R}{dR/d\psi} \tag{1.16}$$

$$\psi \leftarrow \psi + \delta\psi \tag{1.17}$$

Once convergence has been reached, the angle of attack, and flow relative velocity to the blades are known and the lift (L) and drag (D) forces can be obtained per radial station using the airfoil polars. What the airfoil polars are and how they are obtained will be discussed later. The lift and drag forces per radial station (dT and dQ) can be decomposed into thrust and torque as seen in Fig. 1.2 leading to

$$dT = B\frac{1}{2}\rho W^2(cl\ cos\phi - cd\ sin\phi)c\ dr \tag{1.18}$$

$$dQ = B\frac{1}{2}\rho W^2(cl\ sin\phi + cd\ cos\phi)c\ r\ dr, \tag{1.19}$$

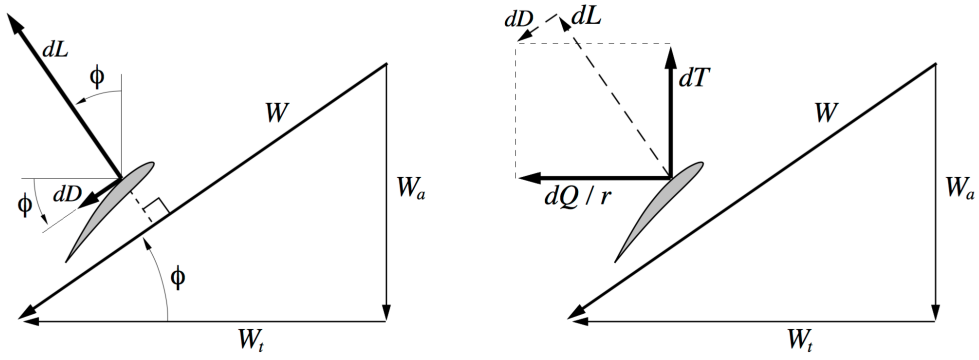

Figure 1.2: Lift and drag decomposition

In equations 1.18 and 1.19 dr is the differential radial corresponding to each elements in which the blade has been discretised.
Drela *[2006]* gives a more detailed description on the theory. The thrust and torque represent the whole contribution at that radial station, and thus, since an actuator disk is going to be simulated, this contribution needs to be equally spread among all the computational cell at that radius. That is why, as it can be seen in Sec. 1.4.5 a cylindrical mesh has been chosen. Once equally spread, the thrust and torque will be decomposed in forces in the x, y and z direction and added to the momentum equation as a source term in the solver.

1.4.3 Airfoil polars

Airfoil polars are going to be a really important thing to take into account when calculating the flow around the propeller, not only because they are use to obtain the lift and drag coefficients, but also because they are a direct consequence of the choice made for the airfoil shapes to use. These curves relate lift and drag coefficients with the blade angle of attack. A typical airfoil polar can be seen in Fig. 1.3.

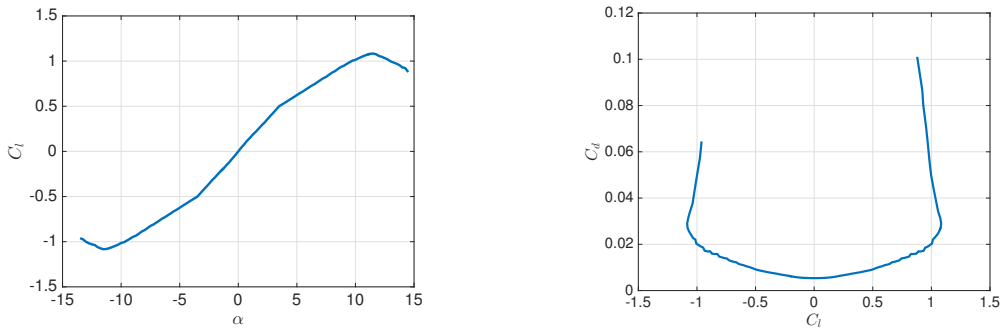

Figure 1.3: Airfoil polars

1.4.4 The Xfoil tool

Xfoil is a program written in Fortran that among other things allows us to generate the polars of a given airfoil and that is released under the GNU General Public License. For purposes of the implementation of the *OpenFOAM* application some small modifications have been made. First download the software by typing

```
cd ~/Desktop
wget http://web.mit.edu/drela/Public/web/xfoil/xfoil6.99.tgz
```

and uncompress it with the command:

```
tar -zxvf xfoil6.99.tgz
rm xfoil6.99.tgz
```

Lets modify a couple of lines in the source code:

```
cd ~/Desktop/Xfoil/src/
```

In this file some lines need to be commented by writing a $'C'$ at the beginning of each line (lines 511 to 518, 534, 548, 550, 554 to 558, 560 to 562, 630 and 631). This can be done by executing:

```
sed -i '511,518{s/^/C/}' iopol.f
sed -i '534{s/^/C/}' iopol.f
sed -i '548{s/^/C/}' iopol.f
sed -i '550{s/^/C/}' iopol.f
sed -i '554,558{s/^/C/}' iopol.f
sed -i '560,562{s/^/C/}' iopol.f
sed -i '630,631{s/^/C/}' iopol.f
```

This will modify the format of the output so it is more suitable for its later use. The compilation instructions need to also to be modified for it to compile (in Chalmers computers). The instructions are:

```
cd ~/Desktop/Xfoil/orrs
```

One line of the file *src/osmap.f* needs to be modified, which can be done as:

```
sed -i s#"/home/codes/orrs/osmapDP.dat"#$PWD/osmapDT.dat#g src/osmap.f
sed -i s#$HOME#"~"#g src/osmap.f
```

In order to comment lines 14 to 17 inclusive, by placing # at the beginning of each line in the file *bin/Makefile*, execute

```
sed -i '14,17{s/^/#/}' bin/Makefile
```

Lets compile part of the code

```
cd ~/Desktop/Xfoil/orrs/bin
make osgen
make osmap.o
```

We still need to modify some more files

```
cd ~/Desktop/Xfoil/plotlib/
sed -i 's+PLTLIB = libPlt_gSP.a+PLTLIB = libPlt_gDP.a+' config.make
make
cd ~/Desktop/Xfoil/bin
sed -i 's+BINDIR = /home/codes/bin/+BINDIR = .+' Makefile_gfortran
```

Finally the applications can be compiled:

```
make -f Makefile_gfortran xfoil
make -f Makefile_gfortran pplot
make -f Makefile_gfortran pxplot
```

If the compilation shows an error such as this one,

```
install: `xfoil' and `./xfoil' are the same file
make: *** [xfoil] Error 1
```

or equivalent for *pplot* or *pxplot*, do not worry, the compilation worked properly. The reason behind this error is that it is trying to move the compiled binaries to the specified directory (*BINDIR*), but we have chosen the current directory. Thus it is trying to move the binaries to the directory where they currently are. For more information on how to use *Xfoil* refer to
`http://web.mit.edu/drela/Public/web/xfoil/`

1.4.5 m4 script for the mesh

In order to be able to modify the mesh easily, an *m4* (*cylindricalMesh.m4*) script has been written to generate the *blockMeshDict* file. In this script,the outer diameter of the cylinder, inner radius, square width and length of the cylinder will be chosen in order to define the geometry (D, Rs, SS and L respectively). On the other hand, in order to define how the mesh will be constructed the following parameters can be modified: *NPS* (the amount of cells along the side of the square), *NPD* (the amount of cells from one circumference to the other in the radial direction), *NPDI* (amount of cells between the square and the smallest circle) and *NPX* (amount of cells in the direction of the free-stream flow). This script has been prepared assuming that the cross sectional topology of the mesh will be as the one shown in Fig. 1.4.

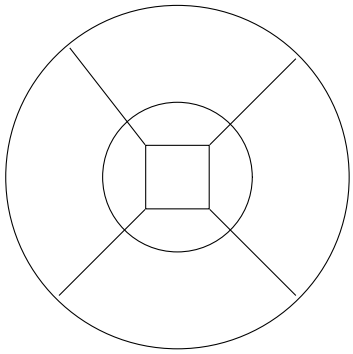

Figure 1.4: Cross sectional topology of the mesh

In order to generate the *blockMeshDict* from the *.m4* file, the following command is executed:

```
m4 cylindricalMesh.m4 > blockMeshDict
```

cylindricalMesh.m4

The script itself is presented here:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.4.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
```

```
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
changecom(//)changequote([,])
define(calc, [esyscmd(perl -e 'printf (\$1)')])
define(VCOUNT, 0)
define(vlabel, [[// ]Vertex \$1 = VCOUNT define(\$1, VCOUNT)define([VCOUNT], incr(VCOUNT))])
convertToMeters 1;
define(D, 6) //column diameter
define(L, 40) // length
define(PI, 3.14159265)
define(R, calc(D/2))
rBig R;
define(Rs, calc(D/15)) //Radius of the smallest section
rSmall Rs;
define(SS, calc(D/70))  // width of half of the square side
halfSquare SS;
define(CW, calc(Rs*cos((PI/180)*45)))
define(CX, calc(R*cos((PI/180)*45)))
define(CZ, calc(R*sin((PI/180)*45)))
define(NPS, 4) //how many cells in the square section
NPSValue NPS;
define(NPD, 20) //how many cells from perimeter to perimeter
NPDValue NPD;
define(NPDI,3)  // how many cells from square to perimeter
NPDIValue NPDI;
define(NPX, 70) // how many cells in X
NPXValue NPX;
vertices
(
    (0.0 CW  CW)
    (0.0 -CW  CW)
    (0.0 -CW -CW)
    (0.0 CW -CW)
    (0.0 CX  CZ)
    (0.0 -CX  CZ)
    (0.0 -CX -CZ)
    (0.0 CX -CZ)
    (L CW  CW)
    (L -CW  CW)
    (L -CW -CW)
    (L CW -CW)
    (L CX  CZ)
    (L -CX  CZ)
    (L -CX -CZ)
    (L CX -CZ)
    (0.0 SS  SS)
    (0.0 -SS  SS)
    (0.0 -SS -SS)
    (0.0 SS -SS)
    (L SS  SS)
    (L -SS  SS)
    (L -SS -SS)
    (L SS -SS)
);
blocks
(
    hex (6 14 10 2 5 13 9 1) (NPX NPD NPS) simpleGrading (2 1 1)
    hex (6 14 15 7 2 10 11 3) (NPX NPS NPD) simpleGrading (2 1 1)
```

```
    hex (3 11 15 7 0 8 12 4) (NPX NPD NPS) simpleGrading (2 1 1)
    hex (1 9 8 0 5 13 12 4) (NPX NPS  NPD) simpleGrading (2 1 1)
    hex (18 22 23 19 17 21 20 16) (NPX NPS NPS) simpleGrading (2 1 1)
    hex (2 10 22 18 1 9 21 17) (NPX NPDI NPS) simpleGrading (2 1 1)
    hex (17 21 20 16 1 9 8 0) (NPX NPS NPDI) simpleGrading (2 1 1)
    hex (19 23 11 3 16 20 8 0) (NPX NPDI NPS) simpleGrading (2 1 1)
    hex (2 10 11 3 18 22 23 19) (NPX NPS  NPDI) simpleGrading (2 1 1)
);
edges
(
    arc 6 5 (0.0 -R 0.0)
    arc 5 4 (0.0 0.0 R)
    arc 7 4 (0.0 R 0.0)
    arc 6 7 (0.0 0.0 -R)
    arc 14 13 (L -R 0.0)
    arc 13 12 (L 0.0 R)
    arc 15 12 (L R 0.0)
    arc 14 15 (L 0.0 -R)
    arc 2 1 (0.0 -Rs 0.0)
    arc 1 0 (0.0 0.0 Rs)
    arc 3 0 (0.0 Rs 0.0)
    arc 2 3 (0.0 0.0 -Rs)
    arc 10 9 (L -Rs 0.0)
    arc 9 8 (L 0.0 Rs)
    arc 11 8 (L Rs 0.0)
    arc 10 11 (L 0.0 -Rs)
);
boundary
(
    inlet
    {
        type patch;
        faces
        (
            (2 6 5 1)
            (0 1 5 4)
            (7 3 0 4)
            (7 6 2 3)
            (19 18 17 16)
            (18 2 1 17)
            (17 1 0 16)
            (19 16 0 3)
            (18 19 3 2)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (14 15 11 10)
            (11 15 12 8)
            (8 12 13 9)
            (14 10 9 13)
            (22 23 20 21)
            (10 11 23 22)
            (10 22 21 9)
            (21 20 8 9)
            (23 11 8 20)
```

```
        );
    }
    walls
    {
        type wall;
        faces
        (
            (6 14 13 5)
            (5 13 12 4)
            (4 12 15 7)
            (15 14 6 7)
        );
    }
);
mergePatchPairs
(
);
```

1.5 Functions defined in *propellerSimpleFoam*

Here is a summary of which functions have been implemented in the solver and what they do:

- ReadGeometry: reads the values under the *propellerData* subdictionary in *fvSolution*. It also calculates the x size of each cell.
- CalcActuatorDiskVolForce: decomposes the thrust and torque calculated in *calculateThrustAndTorque* function and puts them into *VolumeForce* that will be added as a source term in the momentum equation.
- WriteVTK: generated a *.vtk* file that will show the outer surface of the propeller so it can be visualized.
- PointIsInDisk: checks if a given point belong to the actuator disk region or not.
- calculateThrustAndTorque: by means of the equations shown in Sec. 1.4 calculates thrust and torque.
- calculatePolars: if chosen by the user, will call *Xfoil* and generate the polars according to what is found in the *polarsData* subdictionary in *fvSolution*.

1.6 Implementation of propellerSimpleFoam

NOTE: if trying to follow this procedure, be careful with copy pasting. In the *pdf* some lines have been divided in several lines to make them fit in the page width. This should be corrected when implementing the code, if not some errors will appear when compiling. If one line's indent is greater than usual (assuming that the indent is not due to the fact that it is inside a lop or so) should mean that it is a line that has been divided. For example all this piece of code represent one line in the code:

```
prop.CalcActuatorDiskVolForce(mesh, VolumeForce, nps, npd, rIn, rOut, xDimension,
    actuatorDiskThickness, xCoordinateActuatorToPlot, betaVal, chordVal, radiiVal,
    sizeGeo, polarDist, sizeDist);
```

The starting point for the implementation of *propellerSimpleFoam* will be Erik's work that have just been updated to work in *OpenFOAM 2.4x*. We are going to proceed to review all the changes done one by one. In case one wants the final files directly, this can be found under the folder called *propeller* (both the solver and the case). Lets start by copying the *actuatorDiskExplicitForce* and renaming all the files.

```
cd ~/Desktop/gonzaloMonteroFiles/erikSvenningOriginalFiles/
cp -r actuatorDiskExplicitForce/ $WM_PROJECT_USER_DIR/applications/
cd $WM_PROJECT_USER_DIR/applications/
mv actuatorDiskExplicitForce propellerSimpleFoam
cd propellerSimpleFoam
mv actuatorDiskExplicitForceSimpleFoam.C propellerSimpleFoam.C
mv actuatorDiskExplicitForce.cpp propeller.cpp
mv actuatorDiskExplicitForce.h propeller.h
rm *.dep
```

Lets modify the file *Make/files*:

```
sed -i 's+actuatorDiskExplicitForce+propeller+' Make/files
```

With these changes we will make sure that when compiling all the correct files and libraries are used, and also that the name of the solver is as desired, *propellerSimpleFoam*. Lets go ahead with the *propellerSimpleFoam.C* file modifications.

```
sed -i 's+#include "actuatorDiskExplicitForce.h"+#include "propeller.h"+' propellerSimpleFoam.C
```

Right after all the initial *#include* statements introduce this piece of code (after *# include "initContinuityErrs.H"*):

```
//read values from blockMeshDict
fileName polyMeshDir;
IOdictionary meshDict
(
    IOobject
    (
        "polyMesh/blockMeshDict",
        runTime.constant(),
        polyMeshDir,
        runTime,
        IOobject::MUST_READ,
        IOobject::NO_WRITE,
        false
    )
);
scalar nps;
scalar npd;
scalar rIn;
scalar rOut;

ITstream& is1 = meshDict.lookup("NPSValue");
is1.format(IOstream::ASCII);
is1 >> nps;

ITstream& is2 = meshDict.lookup("NPDValue");
is2.format(IOstream::ASCII);
is2 >> npd;

ITstream& is3 = meshDict.lookup("rBig");
is3.format(IOstream::ASCII);
is3 >> rOut;

ITstream& is4 = meshDict.lookup("rSmall");
is4.format(IOstream::ASCII);
is4 >> rIn;

//Initilization of xDimension
volScalarField xDimension
(
    IOobject
    (
        "xDimension",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::NO_WRITE
    ),
    mesh
```

```
);

scalar actuatorDiskThickness;
scalar xCoordinateActuatorToPlot;

//read the distribution of the polars along the radius
std::ifstream fileDist("polarDistribution.txt");
int sizeDist;
fileDist>>sizeDist;
double polarDist[sizeDist];
for (int j=0;j<sizeDist;j++)
{
    fileDist >>polarDist[j];
}
std::ifstream fileGeo("geometry.txt");
int sizeGeo;
fileGeo>>sizeGeo;
double chordVal[sizeGeo];
double betaVal[sizeGeo];
double radiiVal[sizeGeo];
for (int j=0;j<sizeGeo;j++)
{
    fileGeo >>radiiVal[j]>>chordVal[j]>>betaVal[j];
}
```

Here several things are done. First some values from the *blockMeshDict* dictionary are read, and stored for later use. We also initialize a *volScalarField* called *xDimension*, that later will be filled with the length in the x direction each cell. This values will be used later to be able to identify if a cell location belongs to the area where the the actuator disk is placed. Finally we read and store the values from the files *geometry.txt* and *polarDistribution.txt*, which contents will be explained later (in Sec. 1.7.2). Following with the implementation process, lets keep applying changes (still in *propellerSimpleFoam.C*)

```
sed -i 's+actuatorDiskExplicitForce+propeller+' propellerSimpleFoam.C
sed -i 's+actuatorDisk+prop+' propellerSimpleFoam.C
```

Next we will remove

```
//Write geometry to VTK
prop.WriteVTK();
```

and we insert at the end of the file (right before *return (0);*)

```
//Write geometry to VTK
prop.WriteVTK(actuatorDiskThickness, xCoordinateActuatorToPlot);
```

Finally execute:

```
sed -i 's+prop.ReadGeometry(mesh);+prop.ReadGeometry(mesh, xDimension);+' propellerSimpleFoam.C
sed -i 's+scalar propThickness;+scalar actuatorDiskThickness;+' propellerSimpleFoam.C
```

Those are all the changes needed in *propellerSimpleFoam.C*. Now we can move on to make the necessary modifications to the *propeller.h* file. First execute

```
sed -i 's+ACTUATORDISKEXPLICITFORCE_H_+PROPELLER_H_+' propeller.h
sed -i 's+actuatorDiskExplicitForce+propeller+' propeller.h
```

and replace the all the *public:* functions with:

```
void ReadGeometry(const fvMesh &iMesh, volScalarField &xDimension);
void CalcActuatorDiskVolForce(const fvMesh &iMesh, volVectorField &ioVolumeForce,
    const scalar &nps, const scalar &npd, const scalar &rIn, const scalar &rOut,
    volScalarField &xDimension, scalar &actuatorDiskThickness, scalar
    &xCoordinateActuatorToPlot,const double betaVal[],const double chordVal[], const
    double radiiVal[],const int sizeGeo, const double polarDist[], const int sizeDist);

void WriteVTK(scalar &actuatorDiskThickness, scalar &xCoordinateActuatorToPlot);
```

Redefining some function definitions in order to be able to give as an argument all the new variables needed for the computation, such as *mFlightSpeed*. In the *private:* section, lets add the following data members after the ones that are already there (after *scalar mRho;*).

```
scalar mRpm;
scalar mTemperature;
scalar mB; //number of blades
scalar mViscosity;
scalar mReRef;
scalar mReExp;
scalar mDeltaBeta;
scalar mFlightSpeed;
vector mCentrePoint;
```

and remove these one that will not be needed anymore:

```
vector mPointStartCenterLine;
vector mPointEndCenterLine;
scalar mThrust, mTorque;
```

Finally we remove the declarations of the private member functions and we add the following ones:

```
bool PointIsInDisk(const vector &iCentrePoint, const vector &iPoint, scalar &oDist2,
    vector &oCircumferentialDirection, const scalar &xDimensionOfCell);
scalar iterationLoop(const scalar &Ut, const scalar &U, const double alphaVal[],
    const double ClVal[], const int sizePol, const int sizeGeo, const scalar &r,
    const double betaVal[], const double chordVal[], const double radiiVal[],
    const scalar &psi);
void calculateThrustAndTorque(const double alphaVal[], const double ClVal[],
    const double CdVal[], const int sizePol, const int sizeGeo, const scalar &r,
    const double betaVal[], const double chordVal[], const double radiiVal[],
    const scalar &dr, scalar &thrust, scalar &torque);
scalar  interpolate(const double valuesSearchFrom[],
    const double valuesSearchOn[],  const scalar valueInt, const int size);
void calculatePolars(const fvMesh &iMesh);
```

In this last step four new functions have been declared, that will later be explained. Note that the definition for the functions *CalcAxialForce*, *CalcCircForce* and *CalcDiskThickness* have been removed, since in the case of using *propellerSimpleFoam* application, both *CalcAxialForce* and *CalcCircForce* have been replaced by *calculateThrustAndTorque*. This is necessary since the propeller model is different as explained in Sec. 1.4. These are all the changes concerning *propeller.h* file.
Several modifications need to be done in *propeller.cpp* file. First

```
sed -i 's+#include "actuatorDiskExplicitForce.h"+#include "propeller.h" \n #include
    <fstream> \n #include <string>+' propeller.cpp
sed -i 's+actuatorDiskExplicitForce+propeller+' propeller.cpp
sed -i 's+actuatorDiskExplicitForce+propeller+' propeller.cpp
sed -i 's+subDict("actuatorDisk")+subDict("propellerData")+' propeller.cpp
```

The contents in the default constructor should be replace with:

```
mCentrePoint.x() = 0.0;
mCentrePoint.y() = 0.0;
mCentrePoint.z() = 0.0;
mExtRadius = 0.0;
mIntRadius = 0.0;
mRho = 1.225;
mRpm = 0;
mTemperature = 293.15; //in Kelvin
mB = 3; //number of blades
mViscosity = 0.178e-4;
mReRef = 500000;
mReExp = 0.0;
```

The declaration of the function *ReadGeometry* needs to be changed to:

```
void propeller::ReadGeometry(const fvMesh &iMesh, volScalarField &xDimension)
```

At the beginning of the definition of *ReadGeometry*, some variables are read from the *fvSolution* dictionary. There remove:

```
Istream& is3 = iMesh.solutionDict().subDict("actuatorDisk").lookup("thrust");
is3.format(IOstream::ASCII);
is3 >> mThrust;

Istream& is4 = iMesh.solutionDict().subDict("actuatorDisk").lookup("torque");
is4.format(IOstream::ASCII);
is4 >> mTorque;
```

and also remove

```
Istream& is7 = iMesh.solutionDict().subDict("actuatorDisk").lookup("startPoint");
is7.format(IOstream::ASCII);
is7 >> mPointStartCenterLine;

Istream& is8 = iMesh.solutionDict().subDict("actuatorDisk").lookup("endPoint");
is8.format(IOstream::ASCII);
is8 >> mPointEndCenterLine;
```

and add the following lines:

```
Istream& is21 = iMesh.solutionDict().subDict("propellerData").lookup("centrePoint");
is21.format(IOstream::ASCII);
is21 >> mCentrePoint;

Istream& is7 = iMesh.solutionDict().subDict("propellerData").lookup("flightSpeed");
is7.format(IOstream::ASCII);
is7 >> mFlightSpeed;

Istream& is8 = iMesh.solutionDict().subDict("propellerData").lookup("rpm");
is8.format(IOstream::ASCII);
is8 >> mRpm;

Istream& is9 = iMesh.solutionDict().subDict("propellerData").lookup("temperatureKelvin");
is9.format(IOstream::ASCII);
is9 >> mTemperature;

Istream& is10 = iMesh.solutionDict().subDict("propellerData").lookup("numberOfBlades");
is10.format(IOstream::ASCII);
is10 >> mB;

Istream& is11 = iMesh.solutionDict().subDict("propellerData").lookup("dynamicViscosity");
is11.format(IOstream::ASCII);
is11 >> mViscosity;

Istream& is12 = iMesh.solutionDict().subDict("propellerData").lookup("ReRef");
is12.format(IOstream::ASCII);
is12 >> mReRef;

Istream& is13 = iMesh.solutionDict().subDict("propellerData").lookup("ReExp");
is13.format(IOstream::ASCII);
is13 >> mReExp;

Istream& is14 = iMesh.solutionDict().subDict("propellerData").lookup("deltaBeta");
is14.format(IOstream::ASCII);
is14 >> mDeltaBeta;
```

Here some extra variables are read from the *fvSolution* dictionary. The reason why we are using so many new is that we are taking into account the operating condition of the propeller (*rpm*, *temperature* ...). Right after that insert now:

```
const faceList & ff = iMesh.faces();
const pointField & pp = iMesh.points();
scalar xDim;
forAll ( iMesh.C(), celli)
{
    const cell & cc = iMesh.cells()[celli];
    labelList pLabels(cc.labels(ff));
    pointField pLocal(pLabels.size(), vector::zero);

    forAll (pLabels, pointi)
    {
        pLocal[pointi] = pp[pLabels[pointi]];
    }
    xDim = Foam::max(pLocal & vector(1,0,0)) - Foam::min(pLocal & vector(1,0,0));
    xDimension[celli] = xDim;
}

//calcualte polars if needed
calculatePolars(iMesh);
```

and remove:

```
if(debug >= 2) {
    Info << "Actuator disk values loaded from fvSolution:\n";
    Info << "mIntRadius: " << mIntRadius << "\n";
    Info << "mExtRadius: " << mExtRadius << "\n";
    Info << "mThrust: " << mThrust << "\n";
    Info << "mTorque: " << mTorque << "\n";
    Info << "mRho: " << mRho << "\n";
    Info << "mPointStartCenterLine: " << mPointStartCenterLine << "\n";
    Info << "mPointEndCenterLine: " << mPointEndCenterLine << "\n";
}
```

Before, in *propellerSimpleFoam.C* a *volScalarField* has been declared, *xDimension*, and here is where values are assigned to it. In this piece of code every cell's x dimension is calculated and stored in the *volScalarField*. Also the last line calls *calculatePolars* function, which will generate airfoil polars by means of *Xfoil* if required by the user.
Again, another function declaration, in this case the one for *CalcActuatorDiskVolForce* needs to be changed to:

```
void propeller::CalcActuatorDiskVolForce(const fvMesh &iMesh, volVectorField
    &ioVolumeForce, const scalar &nps, const scalar &npd, const scalar &rIn, const scalar &rOut,
    volScalarField &xDimension, scalar &actuatorDiskThickness, scalar &xCoordinateActuatorToPlot,
    const double betaVal[],const double chordVal[],const double radiiVal[],const int
    sizeGeo, const double polarDist[], const int sizeDist){
```

Lets see what needs to be changed inside the definition of *CalcActuatorDiskVolForce*. Right before the for loop (*for(label i = 0; i ¡ iMesh.C()...*)insert this piece of code,

```
scalar TotalForce(0.0);
scalar TotalTorque = 0.0;

int nPolars;
Istream& is=iMesh.solutionDict().subDict("propellerData").lookup("numberOfPolars");
is.format(IOstream::ASCII);
is >> nPolars;
std::ifstream filePol2("polarsData.txt");
```

```
int sizePols[nPolars];
int maxSize = 0;
for (int j=0;j<nPolars;j++)
{
    filePol2 >> sizePols[j];
    if (maxSize < sizePols[j])
    {
        maxSize = sizePols[j];
    }
}
double alphaValAll[maxSize][nPolars];
double ClValAll[maxSize][nPolars];
double CdValAll[maxSize][nPolars];
for (int k=0;k<nPolars;k++)
{
    for (int i=0;i<sizePols[k];i++)
    {
        filePol2 >> alphaValAll[i][k]>>ClValAll[i][k]>>CdValAll[i][k];
    }
}

scalar dr = (rOut-rIn)/npd;
scalar numberOfCellsCirc = 4*nps;
scalar thrust;
scalar torque;
```

we can also remove the line that reads:

```
ReadGeometry(iMesh);
```

and these ones too:

```
vector TotalForce(0.0,0.0,0.0);
scalar TotalTorque = 0.0;
```

Here first we check how many airfoil polars are there, and then each of them is extracted from a *.txt* file, stored and divided into α, C_l and C_d values. This is really of great importance since the lift and the drag forces will aid, not only on the Newton iteration method described in Sec. 1.4 but also because the decomposition of them will be used for the calculation of the thrust and torque (see Eqs. 1.18 and 1.19). Inside the definition of *CalcActuatorDiskVolForce* function, at the beginning of the for loop (*for(label i = 0; i ¡ iMesh.C().size()...*) replace:

```
if(PointIsInDisk(mPointStartCenterLine,mPointEndCenterLine,iMesh.C()[i],RadialDist2, LineTangent,
    CircumferentialDirection)) {
```

with:

```
if(PointIsInDisk(mCentrePoint,iMesh.C()[i],RadialDist2,CircumferentialDirection, xDimension[i])) {
```

inside the *for label i=0...* for loop, inside the *if(PointIsInDisk...* replace:

```
vector axialForce = LineTangent*CalcAxialForce(sqrt(RadialDist2),mRho)/mRho;
ioVolumeForce[i] += axialForce;

//compute the total force added to the actuator disk, this is just for control
TotalForce += axialForce*iMesh.V()[i];

vector circForce = CircumferentialDirection*
    CalcCircForce(sqrt(RadialDist2),mRho)/mRho;
ioVolumeForce[i] += circForce;

TotalTorque += (CalcCircForce(sqrt(RadialDist2),mRho)/mRho)*
    sqrt(RadialDist2)*iMesh.V()[i];
DiskVolume += iMesh.V()[i];
```

with

```
thrust=0;
torque=0;
int exactOrOutsideRangePos=0;
int polarNumberToUse=0;
int firstPolarNumber=0;
if (sqrt(RadialDist2)/mExtRadius<polarDist[0] ||
    sqrt(RadialDist2)/mExtRadius==polarDist[0])
{
    exactOrOutsideRangePos=1;
    polarNumberToUse=0;
}
else if (sqrt(RadialDist2)/mExtRadius>polarDist[sizeDist-1] ||
    sqrt(RadialDist2)/mExtRadius==polarDist[sizeDist-1])
{
    exactOrOutsideRangePos=1;
    polarNumberToUse=sizeDist-1;
}
else
{
    for (int l=0;l<sizeDist-1;l++)
    {
        if (sqrt(RadialDist2)/mExtRadius==polarDist[l])
        {
            exactOrOutsideRangePos=1;
            polarNumberToUse=l;
            break;
        }
        if (sqrt(RadialDist2)/mExtRadius>polarDist[l] &&
            sqrt(RadialDist2)/mExtRadius<polarDist[l+1])
        {
            firstPolarNumber=l;
            exactOrOutsideRangePos=0;
            break;
        }
    }
}
if (exactOrOutsideRangePos==1 || nPolars==1)
{
    if (nPolars==1)
    {
        polarNumberToUse=0;
    }
    double alphaVal[sizePols[polarNumberToUse]];
    double ClVal[sizePols[polarNumberToUse]];
    double CdVal[sizePols[polarNumberToUse]];
    for (int l=0;l<sizePols[polarNumberToUse];l++)
    {
        alphaVal[l]=alphaValAll[l][polarNumberToUse];
        ClVal[l]=ClValAll[l][polarNumberToUse];
        CdVal[l]=CdValAll[l][polarNumberToUse];
    }
    calculateThrustAndTorque(alphaVal, ClVal, CdVal, sizePols[polarNumberToUse],
        sizeGeo,sqrt(RadialDist2),  betaVal, chordVal,radiiVal, dr, thrust,torque);
}
else
{
    scalar thrust1=0;
```

```
    scalar torque1=0;
    double alphaVal1[sizePols[firstPolarNumber]];
    double ClVal1[sizePols[firstPolarNumber]];
    double CdVal1[sizePols[firstPolarNumber]];
    double r1 = polarDist[firstPolarNumber];
    for (int l=0;l<sizePols[firstPolarNumber];l++)
    {
        alphaVal1[l]=alphaValAll[l][firstPolarNumber];
        ClVal1[l]=ClValAll[l][firstPolarNumber];
        CdVal1[l]=CdValAll[l][firstPolarNumber];
    }
    calculateThrustAndTorque(alphaVal1,ClVal1,CdVal1, sizePols[firstPolarNumber],
        sizeGeo, sqrt(RadialDist2), betaVal, chordVal, radiiVal, dr, thrust1, torque1);

    scalar thrust2=0;
    scalar torque2=0;
    double alphaVal2[sizePols[firstPolarNumber+1]];
    double ClVal2[sizePols[firstPolarNumber+1]];
    double CdVal2[sizePols[firstPolarNumber+1]];
    double r2 = polarDist[firstPolarNumber+1];;
    for (int i=0;i<sizePols[firstPolarNumber+1];i++)
    {
        alphaVal2[i]=alphaValAll[i][firstPolarNumber+1];
        ClVal2[i]=ClValAll[i][firstPolarNumber+1];
    CdVal2[i]=CdValAll[i][firstPolarNumber+1];
    }
    calculateThrustAndTorque(alphaVal2,ClVal2,CdVal2, sizePols[firstPolarNumber+1],
        sizeGeo, sqrt(RadialDist2), betaVal, chordVal, radiiVal, dr, thrust2, torque2);
    thrust=(thrust2-thrust1)*(sqrt(RadialDist2)/mExtRadius-r1)/(r2-r1)+thrust1;
    torque=(torque2-torque1)*(sqrt(RadialDist2)/mExtRadius-r1)/(r2-r1)+torque1;
}


vector axialForce = vector(1, 0, 0)*thrust/(numberOfCellsCirc*mRho*iMesh.V()[i]);
ioVolumeForce[i] += axialForce;

TotalForce += thrust/numberOfCellsCirc;

vector circForce = CircumferentialDirection*torque/
    (numberOfCellsCirc*mRho*iMesh.V()[i]*sqrt(RadialDist2));
ioVolumeForce[i] += circForce;

TotalTorque += torque/numberOfCellsCirc;

actuatorDiskThickness = xDimension[i];
xCoordinateActuatorToPlot = iMesh.C()[i].x();
```

With this piece of code, first the actual radial position is evaluated with respect to the ones given in *polarDistribution.txt* to see in which range it is, and therefore which polar, or polars should it use to calculate the thrust and torque. The *polarDistribution.txt* will divide the blade into sections with different airfoil shapes. If it happens that the cell that is being analyzed lays on the radial position where a polar type is defined for that, it will use that one. If it is somewhere in between two of them, linear interpolation will be used. Right before the end of the declaration of *CalcActuatorDiskVolForce* remove:

```
Info << "Total disk volume: " << DiskVolume << "\n";
```

and add:

```
// Some dimensionles numbers
scalar n=mRpm/60;
```

```
scalar diameter=mExtRadius*2;
scalar J=mFlightSpeed/(n*diameter);
scalar Ct=TotalForce/(mRho*pow(n,2)*pow(diameter,4));
scalar Cp=TotalTorque*n*2*3.1415/(mRho*pow(n,3)*pow(diameter,5));
scalar efficiency=Ct*J/Cp;
Info << "Advance ratio: " <<J<<"\n";
Info << "Thrust coefficient: " <<Ct<<"\n";
Info << "Power coefficient: " <<Cp<<"\n";
Info << "Propeller efficiency: " <<efficiency<<"\n\n";
```

Finally once the forces have been computed some typical dimensionless numbers (when working with propellers) are printed in the screen. These dimensionless numbers are defined according to Eqs. 1.20 to 1.23.

$$\text{Advance ratio} \quad \rightarrow \quad J = \frac{V_{freestream}}{n\ D} \tag{1.20}$$

$$\text{Thrust coefficient} \quad \rightarrow \quad C_T = \frac{T}{\rho n^2\ D^4} \tag{1.21}$$

$$\text{Power coefficient} \quad \rightarrow \quad C_P = \frac{P}{\rho n^3\ D^5} \tag{1.22}$$

$$\text{Efficiency} \quad \rightarrow \quad \eta = C_T \frac{J}{C_P} \tag{1.23}$$

where n is the angular velocity of the blades in $[rev/s]$.
The *writeVTK* function declaration needs to be changed to:

```
void propeller::WriteVTK(scalar & actuatorDiskThickness,
    scalar &xCoordinateActuatorToPlot){
```

In the *writeVTK* function definition just below the line that reads:

```
vectorField points(NumPoints,vector::zero);
```

add the following ones

```
vector mPointStartCenterLine = mCentrePoint;
vector mPointEndCenterLine =mCentrePoint;
mPointEndCenterLine.x()=xCoordinateActuatorToPlot+actuatorDiskThickness/2;
mPointStartCenterLine.x()=xCoordinateActuatorToPlot-actuatorDiskThickness/2;
```

Rewrite the whole *PointIsInDisk* function as:

```
bool propeller::PointIsInDisk(const vector &iCentrePoint, const vector &iPoint,
    scalar &oDist2, vector &oCircumferentialDirection,
    const scalar &xDimensionOfCell)
{
////////////////////////////////////////////////////////////////////////
// Check if a given point is located in the actuator disk region.
////////////////////////////////////////////////////////////////////////
    //Is the center of the disk inside the cell boundaries in the axial direction?
    if(!(iCentrePoint.x()>(iPoint.x()-xDimensionOfCell/2) && iCentrePoint.x()<
        (iPoint.x()+xDimensionOfCell/2)))
    {
        return false;
    }
    vector VecLineToPoint(iPoint - iCentrePoint);
    VecLineToPoint.x() = 0.0;
    scalar RadialDist2 = VecLineToPoint.x()*VecLineToPoint.x() + VecLineToPoint.y()
        *VecLineToPoint.y() + VecLineToPoint.z()*VecLineToPoint.z();
    oDist2 = RadialDist2;

    //Check if the point is inside the actuator disk in the radial direction
```

```
    if(!(RadialDist2<=mExtRadius*mExtRadius&& RadialDist2 >= mIntRadius*mIntRadius))
    {
        return false;
    }

    oCircumferentialDirection = vector (1.0, 0.0, 0.0) ^ VecLineToPoint;
    oCircumferentialDirection /= mag(oCircumferentialDirection);
    return true;
}
```

As it name suggest this function will check if a given point is in the actuator disk area. Even though it apparently seems to be the same function as in the case of the *actuatorDiskExplicitForceSimpleForce*, in fact is not. In this application the actuator disk is considered to have the width of one cell whereas in Erik's, the user could choose the width. Thus several change needed to be adjusted in order to account for that. Delete the definition of *PointIsInHub*, *CalcAxialForce* and *CalCircForce* since they are not used anymore. At the end of the file, (before } *//end namespace Foam*) add this pieces of code which contains the first out of four new functions implemented.

```
scalar propeller::interpolate(const double valuesSearchFrom[], const double
    valuesSearchOn[],  const scalar valueInt, const int size)
{
//////////////////////////////////////////////////////////////////////
// Interpolates from two given sets of data
//////////////////////////////////////////////////////////////////////
    scalar interpolatedValue = 0;
    if (valueInt < valuesSearchFrom[0])
    {
        interpolatedValue = valuesSearchOn[0];
    }
    else if (valueInt > valuesSearchFrom[size-1])
    {
        interpolatedValue = valuesSearchOn[size-1];
    }
    else
    {
        for (int i=0;i<size;i++)
        {
            if (valuesSearchFrom[i] == valueInt)
            {
                interpolatedValue = valuesSearchOn[i];
                break;
            }
            else if (valuesSearchFrom[i] < valueInt && valuesSearchFrom[i+1]
                > valueInt && i < size-1)
            {
                interpolatedValue = (valuesSearchOn[i+1]-valuesSearchOn[i])*
                    (valueInt-valuesSearchFrom[i])/(valuesSearchFrom[i+1]-
                    valuesSearchFrom[i])+valuesSearchOn[i];
                break;
            }
        }
    }
    return interpolatedValue;
}
```

As expected this function simply interpolates from two set of values and returns the interpolated value. Note that it is trimmed, which means that if the value that you want to interpolate for, it is outside the range, the closest value (either the first or the last one) will be returned. The second function that needs to be added after the last one reads:

```
scalar propeller::iterationLoop(const scalar &Ut, const scalar &U, const
```

```
    double alphaVal[], const double ClVal[], const int sizePol, const
    int sizeGeo, const scalar &r, const double betaVal[], const double
    chordVal[], const double radiiVal[], const scalar &psi)
//////////////////////////////////////////////////////////////////////
// Runs the sets of equations needed to check if convergence has been reached
//////////////////////////////////////////////////////////////////////
{
    scalar pi = M_PI;
    scalar Wa = 0.5*mFlightSpeed + 0.5*U*sin(psi);
    scalar Wt = 0.5*Ut + 0.5*U*cos(psi);
    scalar vt = Ut- Wt;
    scalar beta = interpolate (radiiVal, betaVal, r/mExtRadius,  sizeGeo)+
        mDeltaBeta;
    scalar alpha = beta - atan(Wa/Wt)*180/pi;
    scalar chord = interpolate (radiiVal, chordVal, r/mExtRadius,
    sizeGeo);
    scalar W = sqrt(pow(Wt,2) + pow(Wa,2));
    scalar a = sqrt(287*1.4*mTemperature);
    scalar Ma = W/a;
    scalar lambda_w = r*Wa/(mExtRadius*Wt);
    scalar f = 0.5*mB*(1-r/mExtRadius)/lambda_w;
    scalar F = 2*acos(exp(-f))/pi;
    scalar Gamma = vt*4*pi*r*F*sqrt(1+pow(4*lambda_w*mExtRadius
        /(pi*mB*r),2))/mB;
    scalar Cl = interpolate (alphaVal, ClVal, alpha, sizePol);
    //applying local Prantdl-Meyer compressibility factor
    Cl = Cl/(sqrt(1 - pow(Ma,2)));
    scalar residual = Gamma - 0.5*W*Cl*chord;
    return residual;
}
```

This function will execute the Newton iteration shown in Sec. 1.4 and will return the residual. Also add this other function:

```
void propeller::calculateThrustAndTorque(const double alphaVal[], const double
    ClVal[], const double CdVal[], const int sizePol, const int sizeGeo, const
    scalar &r, const double betaVal[], const double chordVal[], const double
    radiiVal[], const scalar &dr, scalar &thrust, scalar &torque)
//////////////////////////////////////////////////////////////////////
// Calculates the thrust and torque at a given radial position
//////////////////////////////////////////////////////////////////////
{
    scalar pi = M_PI;
    scalar residual1 = 1;
    scalar residual2 = 0;
    scalar eps = 1e-5;
    scalar newtonConvergence = 1e-6;
    scalar dRdPsi;
    scalar deltaPsi = 0;
    scalar omega = mRpm*pi/30;
    scalar Ut = omega*r;
    scalar U = sqrt(pow(mFlightSpeed,2) + pow(Ut,2));
    scalar psi = 2*atan(mFlightSpeed/Ut);
    while (residual1 > newtonConvergence)
    {
    psi = psi + deltaPsi;
    residual1 = iterationLoop(Ut, U, alphaVal, ClVal, sizePol, sizeGeo, r,
        betaVal, chordVal, radiiVal, psi);
    residual2 = iterationLoop(Ut, U, alphaVal, ClVal, sizePol, sizeGeo, r,
        betaVal, chordVal, radiiVal, psi+eps);
```

```
    dRdPsi = (residual2 - residual1)/eps;
    deltaPsi = -residual1/dRdPsi;
    }
    scalar Wa = 0.5*mFlightSpeed + 0.5*U*sin(psi);
    scalar Wt = 0.5*Ut + 0.5*U*cos(psi);
    scalar beta=interpolate(radiiVal,betaVal,r/mExtRadius, sizeGeo)+mDeltaBeta;
    scalar alpha = beta - atan(Wa/Wt)*180/pi;
    scalar W = sqrt(pow(Wt,2) + pow(Wa,2));
    scalar chord = interpolate (radiiVal, chordVal, r/mExtRadius, sizeGeo);
    scalar Re = mRho*W*chord/mViscosity;
    scalar a = sqrt(287*1.4*mTemperature);
    scalar Ma = W/a;
    scalar Cl = interpolate (alphaVal, ClVal, alpha, sizePol);
    //applying local Prantdl-Meyer compressibility factor
    Cl = Cl/(sqrt(1 - pow(Ma,2)));
    scalar Cd = interpolate (ClVal, CdVal, Cl, sizePol);
    Cd = Cd * pow(Re/mReRef,mReExp); //Reynolds number correction
    scalar phi = atan(Wa/Wt);
    thrust = 0.5*mB*mRho*pow(W,2)*(Cl*cos(phi) - Cd*sin(phi))*chord*dr;
    torque = 0.5*mB*mRho*pow(W,2)*(Cl*sin(phi) + Cd*cos(phi))*chord*r*dr;
}
```

This function will call iteratively to *iterationLoop* until convergence has been reached and then it will compute thrust and torque from Eqs. 1.18 and 1.19. And the last one:

```
void propeller::calculatePolars(const fvMesh &iMesh)
////////////////////////////////////////////////////////////////////////
// If needed will calculate the polars by means of Xfoil
////////////////////////////////////////////////////////////////////////
{
    int nPolarsToGenerate;
    Istream& is15 = iMesh.solutionDict().subDict("polarsData").
        lookup("generatePolars");
    is15.format(IOstream::ASCII);
    is15 >> nPolarsToGenerate;

    if (nPolarsToGenerate>0)
    {
        double a[80][nPolarsToGenerate];
        double cl[80][nPolarsToGenerate];
        double cd[80][nPolarsToGenerate];
        int sizeValues[nPolarsToGenerate];
        for (int i=0;i<nPolarsToGenerate;i++)
        {
            char name1[50];
            sprintf(name1,"polar%i.txt",i+1);
            const char* fileName=name1;

            char inst1[50];
            sprintf(inst1,"instructions%i.txt",i+1);
            const char* instructions=inst1;

            char searchFor1[50];
            sprintf(searchFor1,"type%i",i+1);
            std::string searchForType=string(searchFor1);

            word aa = word(iMesh.solutionDict().subDict("polarsData").
                lookup(searchForType));
            string type1=aa;
            char *type = new char[type1.length() + 1];
```

```
std::strcpy(type,type1.c_str());

char searchFor2[50];
sprintf(searchFor2,"airfoilName%i",i+1);
std::string searchForAirfoil=string(searchFor2);

if (type1 == "naca")
{
    int airfoilType;
    Istream& is17 = iMesh.solutionDict().subDict("polarsData").
        lookup(searchForAirfoil);
    is17.format(IOstream::ASCII);
    is17 >> airfoilType;
    Info<<"Calculating polars for NACA: "<<airfoilType<<"\n";

    FILE *myFile;
    myFile=fopen(instructions,"w");
    fprintf(myFile,"./xfoil\n");
    fprintf(myFile,"plop\n");
    fprintf(myFile,"g\n");
    fprintf(myFile,"\n");
    fprintf(myFile,"naca %i\n",airfoilType);
    fprintf(myFile,"oper\n");
    fprintf(myFile,"iter 70\n");
    fprintf(myFile,"visc 50000\n");
    fprintf(myFile,"pacc\n");
    fprintf(myFile,"%s\n",fileName);
    fprintf(myFile,"\n");
    fprintf(myFile,"aseq -15 15 1\n");
    fprintf(myFile,"\n");
    fprintf(myFile,"quit\n");
    fclose(myFile);
}
else if (type1 == "geometry")
{
    word bb = word(iMesh.solutionDict().subDict("polarsData")
        .lookup(searchForAirfoil));
    string bb1=bb;
    char *geometryName = new char[bb1.length() + 1];
    std::strcpy(geometryName,bb1.c_str());

    Info<<"Calculating polars for the airfoil defined in "
        <<geometryName<<"\n";
    FILE *myFile;
    myFile=fopen(instructions,"w");
    fprintf(myFile,"./xfoil\n");
    fprintf(myFile,"plop\n");
    fprintf(myFile,"g\n");
    fprintf(myFile,"\n");
    fprintf(myFile,"load %s\n",geometryName);
    fprintf(myFile,"mdes\n");
    fprintf(myFile,"filt\n");
    fprintf(myFile,"exec\n");
    fprintf(myFile,"\n");
    fprintf(myFile,"pane\n");
    fprintf(myFile,"oper\n");
    fprintf(myFile,"iter 70\n");
    fprintf(myFile,"visc 50000\n");
    fprintf(myFile,"pacc\n");
```

```
            fprintf(myFile,"%s\n",fileName);
            fprintf(myFile,"\n");
            fprintf(myFile,"aseq -15 15 1\n");
            fprintf(myFile,"\n");
            fprintf(myFile,"quit\n");
            fclose(myFile);
        }
        else
        {
            Info<<"WARNING: in the polar number "<<i<<
            " the type is not valid, choose between naca or geometry \n";
        }
        char command[80];
        sprintf(command,"./xfoil < %s 1>/dev/null",instructions);
        system(command);
        char command1[80];
        sprintf(command1,"rm %s",instructions);
        system(command1);
        double iy;
        double it;
        double ir;
        double ie;
        double iw;
        double iq;
        int size=0;
        std::ifstream file;
        file.open(fileName);
        if (file.is_open())
        {
            while (!file.eof())
            {
                file >> a[size][i]>>cl[size][i]>>cd[size][i]>>iy>>it>>
                    ir>>ie>>iw>>iq;
                size=size+1;
            }
        }
        file.close();
        char command2[80];
        sprintf(command2,"rm %s",fileName);
        system(command2);
        size=size-1;
        sizeValues[i]=size;
    }
    FILE *myFileP;
    myFileP=fopen("polarsData.txt","w");
    for (int go=0;go<nPolarsToGenerate;go++)
    {
        fprintf(myFileP,"%i\n",sizeValues[go]);
    }
    for (int po=0;po<nPolarsToGenerate;po++)
    {
        for (int el=0;el<sizeValues[po];el++)
        {
            fprintf(myFileP,"%5.2f %6.4f %6.4f\n",a[el][po],
                cl[el][po],cd[el][po]);
        }
    }
    fclose(myFileP);
}
```

```
}
```

Finally, this function will be in charge of automatizing the process of obtaining the airfoil polars with *Xfoil* if the users decides to do so. It can generate polars both from a NACA 4 or 5 type, or in a more general case, from the geometry of the airfoil given in a *.txt*. This *.txt* file needs to have some specific format and ordering of points. More about this can be read in it's original user guide:
`http://web.mit.edu/drela/Public/web/xfoil/xfoil_doc.txt`
To end up with the required modifications, we have to go to the *UEqn.H* file. Here simply replace the line that calls to *actuatorDisk.CalcActuatorDiskVolForce* with:

```
prop.CalcActuatorDiskVolForce(mesh, VolumeForce, nps, npd, rIn, rOut, xDimension,
    actuatorDiskThickness, xCoordinateActuatorToPlot, betaVal, chordVal, radiiVal,
    sizeGeo, polarDist, sizeDist);
```

Now it is ready to be compiled, just type when being in propellerSimpleFoam directory:

```
wmake
```

1.7 How to define the propeller

In order to define the propellers geometry and operating conditions several data inputs are needed. The required data will be provided by means of the *fvSolution* dictionary and using some *.txt* files.

1.7.1 The *fvSolution* dictionary

The information provided here, concerns the operating conditions and some basic geometrical data. In this dictionary the following is needed, and will be divided into two subdictionaries. The first one looks like this:

```
propellerData
{
    numberOfPolars      10;
    flightSpeed         30;
    deltaBeta           30;
    centrePoint         (15 0 0);
    density             0.36518;
    interiorRadius      0.49;
    exteriorRadius      1.524;
    rpm                 500;
    temperatureKelvin   216.86;
    numberOfBlades      3;
    dynamicViscosity    0.1433e-4;
    ReRef               50000;
    ReExp               0;
}
```

Lets explain some of the terms that might be confusing or not completely clear.

- numberOfPolars: this will determine how many different polars will be needed to compute the forces in the propeller. It is not really common to have a propeller blade with constant airfoil shape along the radius, normally the shape of the aerofoil varies with the radius as can be inferred from Fig. 1.5. Thus, the user can choose the amount of polars that are needed to better represent the problem to simulate. This will be done via the *polarsData* sub dictionary and the file *polarsData.txt* that later will be described.

- deltaBeta: Propeller may vary the pitch angle of their blades by rotating them. This *deltaBeta* represent that offset that will be given to the whole blade (value in degrees). A representation of what the pitch angle is can be seen in Fig. 1.6.

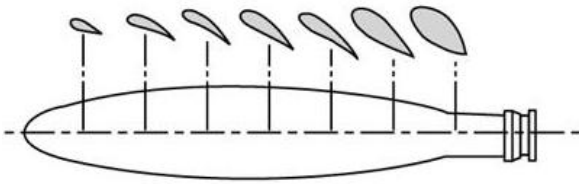

Figure 1.5: Cross sectional airfoils at different radial stations

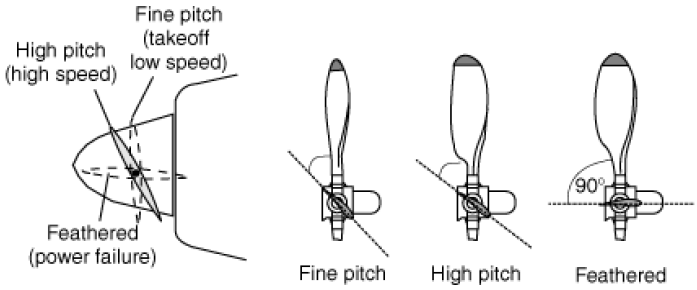

Figure 1.6: Ilustration of the pitch angle

- ReRef and ReExp: The value of *ReRef* is the Reynolds number at which the polars that will be used were obtained. In case that the polars are generated by means of *Xfoil*, this value should be set to 50000. *ReExp* will be use to correct the C_d obtained from the polar corresponding to the reference Reynolds number. The correction will be done as follows:

$$C_{d,corrected} = C_{d,polars}\left(\frac{Re}{ReRef}\right)^{ReExp} \tag{1.24}$$

  where Re is the actual Reynolds number at the operating conditions. The lift coefficient, C_l, is also corrected to try to account for compressibility effects. This is done via Prandtl-Meyer compressibility factor as:

$$C_{l,corrected} = \frac{C_{l,polars}}{\sqrt{1 - Ma^2}} \tag{1.25}$$

  where Ma is the Mach number.

The second subdictionary will contain all the information needed in case the polars needs to be generated with *Xfoil*.

```
polarsData
{
    generatePolars  2;

    type1           naca;
    airfoilName1    2412;

    type2           geometry;
    airfoilName2    ARA-D6.txt;
}
```

- generatePolars: This is going to be the parameter that indicates if *Xfoil* will be used for generating the polars or they will be provided by the user in a file named *polarsData.txt* which format and content will be later discussed. If *generatePolars* is equal to 0, it means that the user supplies the polars, if different, *Xfoil* will generate that amount of polars by taking into account the other two keywords in the subdictionary. Note that if automatic polars generation is used, the value of *generatePolars* and *numberOfPolars* in the *propellerData* subdirectory need to be the same, as well as the amount of radial station given in *polarsData.txt*.
- type: This parameters can be either *naca* or *geometry*. The number after *type* indicates that those values will be used to generate that polar number.

- airfoilName: If *naca* is choosen in it's correspondent *type*, a naca 4 or 5 must be provided and *Xfoil* will generate the geometry first and then the polars. If instead *geometry* has been chosen, the name of the file containing $2D$ coordinates of the airfoil shape must be provided, as in the example subdictionary. The format of this file can be found in *Xfoil* documentation
`http://web.mit.edu/drela/Public/web/xfoil/xfoil_doc.txt`

1.7.2 The .txt files

There are three different files that might be used when running the case. These are, *geometry.txt* and *polarDistribution.txt*, which will always be required, and the third one which is *polarsData.txt* that will be required when the polars are not automatically generated. Let's go through the specific format that they should have.

- *geometry.txt*: In this file the geometry of the propeller will be defined. Data regarding chord and pitch angle, (β), is provided for different radial stations. Here is an example:

```
10
0.3220  0.0568  15.8540
0.3626  0.0634  13.8499
0.3934  0.0682  12.1941
0.4450  0.0738  9.3355
0.5076  0.0761  6.3916
0.6230  0.0717  2.7624
0.7349  0.0625  0.4000
0.8468  0.0510  -1.4360
0.9586  0.0365  -3.0595
0.9900  0.0325  -3.5147
```

  The first number (in this case 10) indicates the number of rows of the table (without itself been taken into account). Each row represents one radial station. The first column is value of the radius at that radial station divided by the tip radius of the propeller, $\frac{r}{R_{tip}}$. The second column contains the values of the blade chord (in meters) at those locations indicated by column one, and the third column represent the value of β (in degrees).

- *polarsData.txt*: This file is only needed if *Xfoil* is not used. In order to show how the format looks like lets use a commented example:

```
2                       //number of rows given for polar 1
3                       //number of rows given for polar 2
2                       //number of rows given for polar 3
-2  0.6648  0.0314      //point 1 of polar number 1
0   0.7890  0.0422      //point 2 of polar number 1
-5  0.3544  0.0160      //point 1 of polar number 2
-1  0.7131  0.0287      //point 2 of polar number 2
6   1.0977  0.0160      //point 3 of polar number 2
2   0.8188  0.0203      //point 1 of polar number 3
7   1.2003  0.0330      //point 2 of polar number 3
```

  The first column represents the value of the angle of attack (in degrees), the second one is C_l and the third one is C_d.

- *polarDistribution.txt*: In this file the information about which airfoil polar corresponds to each radial station is stored. As an example:

```
10
0.382
0.3926
0.4334
0.445
0.5376
0.613
0.724
0.8558
```

```
0.9516
0.99
```

The first number, once again, indicates the number of rows (without itself been taken into account). The values that are stored in this file are $\frac{r}{R_{tip}}$. What every value represents, is the association of that radial station with the corresponding polar. For the file shown above for example:

$$\frac{r}{R_{tip}} = 0.382 \quad \rightarrow \quad polar1$$

$$\frac{r}{R_{tip}} = 0.3926 \quad \rightarrow \quad polar2$$

and so on. For radial stations that lie in between when going through all the cells, linear interpolation will be used. Once again this first number has to match the rest in *fvSolution* dictionary regarding the amount of polars.

1.8 Setting up a case

Before this, make sure that *propellerSimpleFoam* is compiled. The final files have been included in the accompanying files (*gonzaloMonteroFiles/propeller/propellerCase/*). Lets start once again from Erik's case and by copying (into the *$FOAM_RUN* directory) the *cylindricalMesh.m4* file presented in Sec. 1.4.5, and the three *.txt* that will be needed later.

```
cd ~/Desktop/gonzaloMonteroFiles/erikSvenningOriginalFiles/
cp -r caviyActuatorDisk $FOAM_RUN
cd ../Additional\ files/
cp cylindricalMesh.m4 $FOAM_RUN/caviyActuatorDisk/constant/polyMesh
cp *.txt $FOAM_RUN/caviyActuatorDisk/
```

Note that the three binaries (*pxplot*, *pplot* and *xfoil*) generated when compiling *Xfoil* should be place in the same folder as the *.txt* files.

```
cd ~/Desktop/Xfoil/bin/
cp pxplot xfoil pplot $FOAM_RUN/caviyActuatorDisk/
cd $FOAM_RUN
mv caviyActuatorDisk propellerCase
cd propellerCase/constant/polyMesh
gedit cylindricalMesh.m4
```

Now lets modify *cylindricalMesh.m4* to customize our mesh and computational domain, for example, by setting the following values:

- D=5
- L=50
- modify the definition of Rs, to get:

  ```
  define(Rs, calc(D/13))
  ```

- NPS=6
- NPD=20
- NPDI=4
- NPX=40

NOTE: it is important that the actuator disk area lays in between the two circular sections.
Lets execute the *.m4* file to generate the *blockMeshDict* with the wanted mesh.

```
m4 cylindricalMesh.m4 > blockMeshDict
cd ../..
blockMesh
cd 0
gedit U
```

We will set up the U file like this:

```
internalField   uniform (15 0 0);

boundaryField
{
    inlet
    {
        type    zeroGradient;
    }
    outlet
    {
        type    zeroGradient;
    }
    walls
    {
        type    slip;
    }
}

gedit p
```

We have set the uniform velocity to 15 m/s (always axial direction), so later in the dictionary we will have to be consistent with it. We will set it up like this the file for the pressure:

```
internalField   uniform 0;

boundaryField
{
    inlet
    {
        type    zeroGradient;
    }
    outlet
    {
        type    zeroGradient;
    }
    walls
    {
        type    zeroGradient;
    }
}
```

We have to initialize the *volVectorField VolumeForce*. For that lets make a copy of the *U* file and modify it.

```
cp U VolumeForce
gedit VolumeForce
```

Change the dimensions to [0 1 $-$2 0 0 0 0] and the object to *VolumeForce*. The rest will be set as:

```
dimensions      [0 1 -2 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type        fixedValue;
        value       uniform (0.0 0.0 0.0);
    }
```

```
    outlet
    {
        type        fixedValue;
        value       uniform (0.0 0.0 0.0);
    }
    walls
    {
        type        fixedValue;
        value       uniform (0.0 0.0 0.0);
    }
}
```

Now we need to create another one for the *volScalarField xDimension*, so since it is a scalar field this time we will copy and modify the pressure file:

```
cp p xDimension
gedit xDimension
```

Once again remember to change the dimension, in this case to meters and also to change the object field to *xDimension*. Finally it should be like this one:

```
dimensions      [0 1 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type        fixedValue;
        value       uniform 0;
    }
    outlet
    {
        type        fixedValue;
        value       uniform 0;
    }
    walls
    {
        type        fixedValue;
        value       uniform 0;
    }
}
```

Now lets go on and modify the *fvSolution* dictionary.

```
cd ../system/
gedit fvSolution
```

In the *SIMPLE* subdictionary add:

```
pRefCell        0;
pRefValue       0;
```

Rename the *actuatorDisk* as *propellerData*. Add the missing fields so that it ends up looking like this:

```
propellerData
{
    numberOfPolars      2;
    flightSpeed         15;
    deltaBeta           30;
    centrePoint         (15 0 0);
```

```
    density             0.36518;
    interiorRadius      0.49;
    exteriorRadius      1.524;
    rpm                 500;
    temperatureKelvin   216.86;
    numberOfBlades      3;
    dynamicViscosity    0.1433e-4;
    ReRef               50000;
    ReExp               0;
}
```

Those values of density, viscosity, temperature... have been obtained assuming flight altitude of 36000 ft and using *ISA* tables. According to those, the value of *nu* should be also replaced in *transportProperties* file. In *fvSolution* create a subdictionary with the following structure and named *polarsData*.

```
polarsData
{
    generatePolars  2;

    type1           naca;
    airfoilName1    2412;

    type2           geometry;
    airfoilName2    NACA0012.txt;
}
```

```
gedit fvSchemes
```

and add the following change the divergence schemes so they look like:

```
default                         none;
div(phi,U)                      bounded Gauss upwind;
div(phi,k)                      bounded Gauss upwind;
div(phi,epsilon)                bounded Gauss upwind;
div(phi,R)                      Gauss upwind;
div(R)                          Gauss linear;
div(phi,nuTilda)                Gauss upwind;
div((nuEff*dev(grad(U).T())))   Gauss linear;
div((nuEff*dev(T(grad(U)))))    Gauss linear;
```

Now we are ready to run the case:

```
cd ..
propellerSimpleFoam
```

NOTE: it may happen that when trying to calculate the polars of the airfoil by means of *Xfoil* we may see this message on the terminal:

```
sh: ./xfoil: Permission denied
rm: cannot remove `polar1.txt': No such file or directory
```

If so, simply remove the downloaded binaries of *xfoil*, *pxplot* and *pplot*, compile them yourself, and copy them to the same directory. This should fix the problem.

1.9 Some results and validation

Some simulations have been run and some sort of validation has carried out. Results obtained by [Hartman and Bierman, 1938] have been tried to reproduce. More precisely, Fig. 9, in the mentioned report, was the curve used and the result can be seen in Fig. 1.7. Note that the accuracy may not be very good since in order to obtain data from the graphs in the report, Plotdigitizer software was used.
Finally some contours and streamlines to visualize the flow around the propeller in Fig. 1.8.

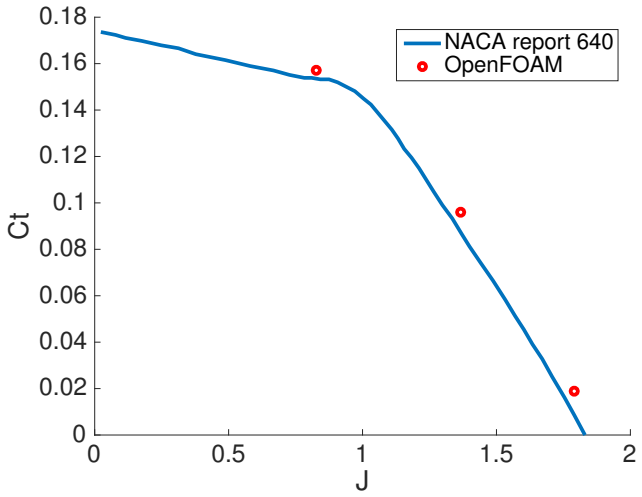

Figure 1.7: Validation of model with NACA report 640, Fig. 9, 35 degree at 0.75R

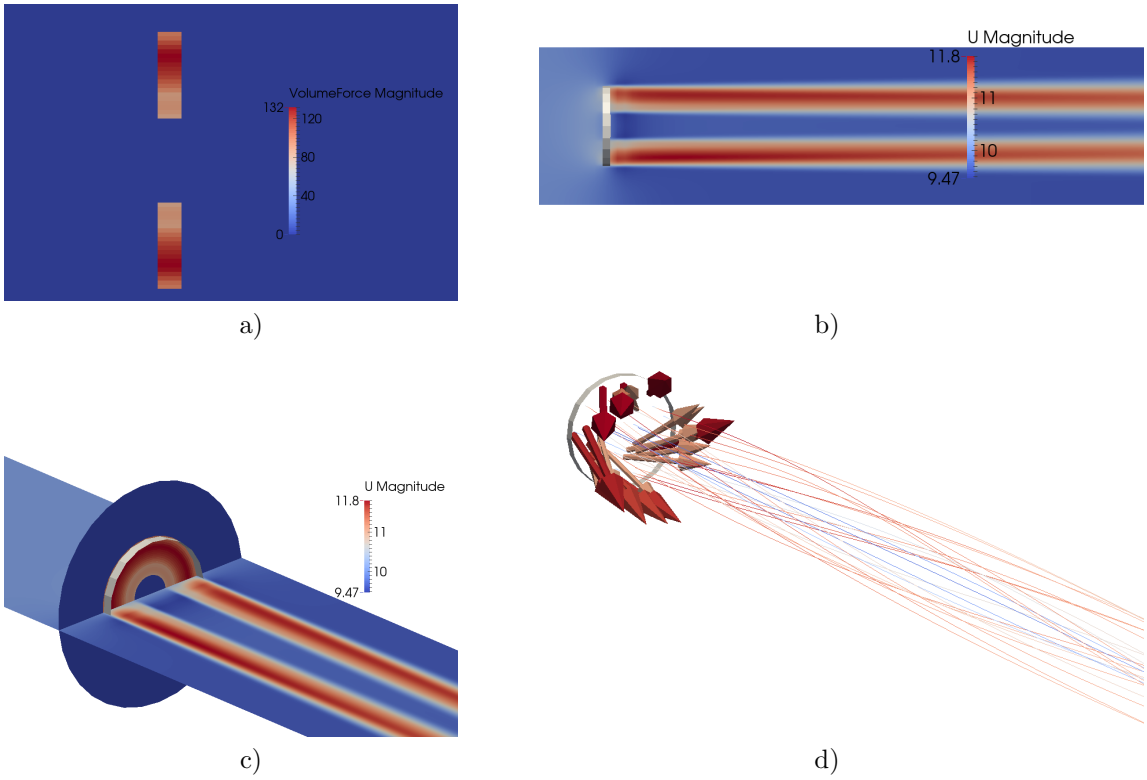

Figure 1.8: Flow visualization around the propeller, a) volume force in the area of the propeller, b) velocity magnitude contour, c) velocity magnitude contour and volume force contour on the disk and d) streamlines and column force vectors

1.10 Study questions

- What do we use the propellerSimpleFoam for?
- What is the main difference with respect to the project carried out by Erik Sven- ning in 2010?
- What are the polars of an airfoil and how do we use them here?
- How do we relate the obtained thrust and torque with the fluid?
- What is Xfoil and how do we use it here?
- How can we execute commands on the terminal while running OpenFoam (C++ code) and how can this be useful for us?
- How is the .m4 used here?

Bibliography

[Hartman and Bierman, 1938] Hartman, E. P. and Bierman, D. (1938). The aerodynamic characteristics of full-scale propellers having 2, 3 and 4 blades of clark y and r.a.f. 6 airfoil sections. naca report no. 640.

[Svenning, 2010] Svenning, E. (2010). Implementation of an actuator disk in OpenFOAM.