Adding a simple body force propeller model to simpleFoam

By: Björn Windén

January 26, 2022
Source : http://ocean-cfd.engr.tamu.edu/doc/bodyforcesimple

Attachment : propellerTutorialCase.zip

Report No.: **OCEN CFD Technical Report #220002**

1 Introduction

This document describes the steps needed to implement a simple body force propeller model in OpenFOAM. Body forces are a way to represent objects interfering with a fluid without discretizing said object. Consider a body moving in a fluid. This body will disturb the surrounding fluid by transferring some of its momentum through viscous and normal stresses. This can be approximated by introducing a forcing term F_v in the Navier-Stokes momentum equation as shown in Eq. (1). A body force model of the moving object is a formulation for the distribution of F_v in the fluid; such that the generated momentum is accurately represented.

$$
\begin{aligned}
\frac{\partial(\rho u)}{\partial t} + \nabla\cdot(\rho u\overline{u}) &= -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho F_{vx} \\
\frac{\partial(\rho v)}{\partial t} + \nabla\cdot(\rho v\overline{u}) &= -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho F_{vy} \\
\frac{\partial(\rho w)}{\partial t} + \nabla\cdot(\rho w\overline{u}) &= -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho F_{vz}
\end{aligned}
\tag{1}
$$

In this case, let's consider the momentum introduced by a rotating propeller. Approximating this using a body force model presents a major advantage in ship performance calculations. Discretizing a complex rotating geometry behind an already complex geometry (ship's hull + appendages) is both challenging and computationally intensive.

There are many approaches to describing a propeller using a distribution of F_v. For example, the open source self propulsion framework SHORTCUt has several formulations implemented, based on Blade Element Momentum theory and Lifting Line/Surface theory. See Windén (2021a,b) for more information. In this report, a very simple model will be created to demonstrate the principle of how to modify OpenFOAM solvers for this kind of purpose. These are the goals that will be addressed in the following sections:

- The model should be able to read basic propeller geometry from a dictionary during run-time
- The momentum source strength should be based on the generated thrust and torque from the propeller. As a simple example, the thrust and torque will be obtained from a curve-fit to experimental data for a given propeller advance ratio
- The advance ratio should be obtained by probing the propeller inflow velocity

2 Preparaion

First, let's create a directory where you will keep the source code of the new solver.

```
mkdir my_applications
cd my_applications
```

Now, let's get a copy of simleFoam and call it my_simpleFoam. Note the path to the root installation of OpenFOAM; if yours is different then adjust accordingly.

```
cp -r /opt/OpenFOAM/OpenFOAM-7/applications/solvers/incompressible/simpleFoam/
    ↪ .
mv simpleFoam my_simpleFoam
cd my_simpleFoam
mv simpleFoam.C my_simpleFoam.C
```

Remove derivative solvers we don't need.

```
rm -r porousSimpleFoam
rm -r SRFSimpleFoam
```

Finally, before we start modifying the source code, let's modify the make files. We need to change the name and location of the target executable to avoid overwriting anything.

```
sed -i "s/simpleFoam/my_simpleFoam/g" Make/files
sed -i "s/FOAM_APPBIN/FOAM_USER_APPBIN/" Make/files
```

3 Creating a body force field variable

Next, we need to have the solver create/read the body force field. In *createFields.H*, add the following just under the intitialization of U.

```
Info<< "Reading field volumeForce\n" << endl;
volVectorField volumeForce               //C++ name "volumeForce"
(
    IOobject
    (
        "volumeForce",           //Written file name "volumeForce"
        runTime.timeName(),      //Written in the time folder (e.g. 0,1,2,3...)
        mesh,                    //Stored in the mesh database
        IOobject::MUST_READ,     //Must be read if the file exists (e.g. when
            ↪ restarting a simulation from a time other than 0)
        IOobject::AUTO_WRITE     //Automatically write at all time steps where
            ↪ writing is requested by controlDict
    ),
    mesh
);
```

The boundary conditions of the *volumeForce* field are defined by adding a file called *volumeForce* in the 0-directory in any case that is to use the my_simpleFoam solver. The *volumeForce* file should contain the following:

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
```

```
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      volumeForce;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform (0 0 0);

boundaryField
{
    ".*"
    {
        type            zeroGradient;
    }

}
```

which sets the boundary condition of F_v to *zeroGradient* for all boundaries.

4 Adding the body force in the solver

my_simpleFoam.C is the source code for the solver. We need to add an expression for the volume force here. Add the following above `#include "Ueqn.H"`

```
#include "forceEqn.H"
```

Also in *my_simpleFoam.C*, add the following below `#include "initContinuityErrs.H"`

```
#include "initProp.H"
```

Finally, we need to add the body force field to the Navier Stokes momentum equation. In *Ueqn.H*, Add the volume force to the right hand side of the equation, modifying the definition of `tmp<fvVectorMatrix> tUEqn` as:

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

5 Initializing the propeller model from a dictionary

In Section 4, *initProp.H* was added to the simpleFoam source code to initialize the propeller model. Here, we define how the solver should read propeller properties from a dictionary. Let's define a new dictionary, it should be called *propellerDict* and be placed in the *system*-folder of a case where you would like to use *my_simpleFoam*. In this example, the propeller thrust and torque will be obtained by curve fit to an experimental open water curve. The dictionary therefore contains 5th order polynomial coefficients that describes the relationship between advance ratio and thrust/torque for a MARIN 7967 propeller. The dictionary can be modified with different coefficients to fit any other available open water curves.

propellerDict should contain the following information:

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  7
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      propellerDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

propOrigin       (6 0 0);
propOrientation (1 0 0);
propVertDir (0 0 1);

radius          1;
hubRadius       0.2;
thickness       0.2;

n               20;
frontUd         4;

//J vs KT,KQ polynomial coefficients for MARIN 7967 propeller a0+a1*J+a2*J^2+a3
   ↪ *J^3+a4*J^4+a5*J^5
KTfifthOrderPolyCoeffs  (0.398399 -0.067794 -1.286040 2.286960 -2.039820
   ↪ 0.676134);
KQfifthOrderPolyCoeffs  (0.051144 -0.000390 -0.171650 0.330060 -0.327865
   ↪ 0.119477);
```

Now we need to set up the solver to read this dictionary. Create a new file called *initProp.H* in the same directory as *my_simpleFoam.C* and initialize the propeller model as follows:

```
//Read the propeller dictionary from the system folder
IOdictionary propDict
(
        IOobject
        (
            "propellerDict",
            runTime.system(),
            runTime,
            IOobject::MUST_READ,
            IOobject::NO_WRITE,
            false
        )
);
```

```
//Read properties from the dictionary
vector propOrigin, propOrient, propVertDir;
propDict.lookup("propOrigin") >> propOrigin;
propDict.lookup("propOrientation") >> propOrient;
propDict.lookup("propVertDir") >> propVertDir;

scalar propRad, hubRad, thickness, n, frontUd;
propDict.lookup("radius") >> propRad;
propDict.lookup("hubRadius") >> hubRad;
propDict.lookup("thickness") >> thickness;
propDict.lookup("n") >> n;
propDict.lookup("frontUd") >> frontUd;

//J vs KT,KQ polynomial fit for propeller KT=a0+a1*J+a2*J^2+a3*J^3+a4*J^4+a5*J
    ↪ ^5
List<scalar> fithOrderCoeffsKT, fithOrderCoeffsKQ;
fithOrderCoeffsKT.setSize(6);
fithOrderCoeffsKQ.setSize(6);
propDict.lookup("KTfifthOrderPolyCoeffs") >>  fithOrderCoeffsKT;
propDict.lookup("KQfifthOrderPolyCoeffs") >>  fithOrderCoeffsKQ;

//Pre-declare some variables
vector tanDir, projPplane, cellC, projPaxis;
scalar currRad, Thrust, Torque, axF, tanF, totvFA, totvFT, cellV, radiusScaled,
    ↪  volinside(0), U0, J, KT, KQ;
DynamicList<label> cellsinside;
DynamicList<vector> cellradii;
int ninside;
label probeCell(-1);

//Make axial vector unit length
propOrient = propOrient/mag(propOrient);
```

6 Defining the body force strength

Let $[U]_{x_{pp}}$ be the total velocity (u, v, w) at a probe location x_{pp}, a distance d_p upstream of the propeller origin x_{p0} at a radius of $0.5r_0$. Also, let the propeller orientation be defined by the propeller disk normal vector $\bar{P}$ and the propeller vertical direction be defined by the vector $\bar{P}_v$. Then

$$x_{pp} = x_{p0} - d_p\bar{P} + 0.5r_0\bar{P}_v \tag{2}$$

and the propeller inflow velocity U_0 can be calculated as

$$U_0 = [U]_{x_{pp}} \cdot P \tag{3}$$

From this, an approximate propeller advance ratio J can be calculated as

$$J = \frac{U_0}{n2r_0}; \tag{4}$$

where n is the propeller rotation rate and r_0 is the propeller radius. Using a 5th order polynomial fit to experimental open water curves, the coefficient of thrust K_T and torque K_Q can be calculated from this advance ratio as

$$K_T = a_0 + a_1J + a_2J^2 + a_3J^3 + a_4J^4 \tag{5}$$

$$K_Q = b_0 + b_1 J + b_2 J^2 + b_3 J^3 + b_4 J^4 \tag{6}$$

Here, the propeller origin x_{p0}, the propeller orientation $\bar{P}$, the distance to the probe point d_p, the propeller radius r_0, the rotation rate n as well as the 5th order coefficients a_{0-4} and b_{0-4} are all read from the propeller dictionary at run-time (as shown in the source code for *initProp.H*) and do not need to be hard-coded into the solver.

From Eq. (5) and Eq. (6), the total thrust T and torque Q of the propeller are given as

$$T = K_T n^2 \left(2r_0\right)^4 \tag{7}$$

$$Q = K_Q n^2 \left(2r_0\right)^5 \tag{8}$$

This methodology yields an approximation of the total forcing to apply. It does not however, model the distribution of the force on the propeller disk. To do this, and thereby obtaining the distribution of F_v on the Finite Volume mesh, a Goldstein (1929) optimal distribution can be used. A non-dimensional radius r_s is defined as

$$r_s = \frac{|R_I| - r_H}{r_0 - r_H} \tag{9}$$

where r_H is the propeller hub radius and R_I is a vector from the propeller origin to the cell center of cell I, projected onto the propeller plane. Two shape functions, f_K and f_Q, describe the distribution of thrust and torque respectively along the blade radius. These are defined, based on r_s, as:

$$f_K = r_s \sqrt{1 - r_s} \tag{10}$$

$$f_Q = \frac{r_s \sqrt{1 - r_s}}{r_s \left(1 - r_H\right) + r_H} \tag{11}$$

The distribution of F_v on the Finite Volume mesh can be obtained by applying the shape functions f_K and f_Q to each cell of the mesh, with $F_v = 0$ being applied outside of the propeller radius and inside the hub radius. The shape functions are based on the cell center location from Eq. (9).

Finally, to ensure consistensy with the calculated thrust and torque from Eq. (7) and Eq. (8), the distribution of F_v is first normalized by the sum of the forces over all cells inside the propeller disk as:

$$F_K = \sum_{cellI} f_K V_I \tag{12}$$

$$F_Q = \sum_{cellI} f_Q V_I r_I \tag{13}$$

where V_I is the volume of cell I. Finally, the strength of F_v in cell I, denoted F_{vI}, is calculated by multiplying the normalized force distribution by the thrust, torque and the appropriate directional vectors.

$$F_{vI} = \frac{\bar{P} T f_K}{F_K} + \frac{\left(\bar{P} \times R_I\right) Q f_Q}{F_Q} \tag{14}$$

This approach is very crude since only one velocity is probed and no consideration is given to the blade geometry (Goldstein optimum used instead.) This means that the method is unable to detect variations in both the wake and propeller geometry and therefore, is not suitable for studying propeller-hull interaction. It is provided here only as a simple example of how a method like this can be implemented. The normalization done in Eq. (14) is consistent with what would be done for more advanced models where f_K, f_Q, T and Q are calculated from the propeller geometry rather than a fixed distribution.

In Section 4, *forceEqn.H* was added to the simpleFoam source code to calculate the distribution of F_v. Create a new file called *forceEqn.H* in the same directory as *my_simpleFoam.C* and calculate the body force as follows:

```
//If the probe cell wasn't searched for yet, find it. This if-statement means
//the mesh only has to be searched once.
if (probeCell == -1)
{
        probeCell = mesh.findCell(propOrigin+propVertDir*0.5*propRad-frontUd*
           ↪ propOrient);
}


//Inflow velocity normal to the propeller
U0=U[probeCell] & propOrient;

//Advance coefficient
J = U0/(n*2*propRad);

//KT and KQ from polynomial fit
KT = fithOrderCoeffsKT[0]+ fithOrderCoeffsKT[1]*J + fithOrderCoeffsKT[2]*sqr(J)
   ↪  + fithOrderCoeffsKT[3]*J*sqr(J) +fithOrderCoeffsKT[4]*sqr(J)*sqr(J) +
   ↪ fithOrderCoeffsKT[5]*J*sqr(J)*sqr(J);
KQ = fithOrderCoeffsKQ[0]+ fithOrderCoeffsKQ[1]*J + fithOrderCoeffsKQ[2]*sqr(J)
   ↪  + fithOrderCoeffsKQ[3]*J*sqr(J) +fithOrderCoeffsKQ[4]*sqr(J)*sqr(J) +
   ↪ fithOrderCoeffsKQ[5]*J*sqr(J)*sqr(J);

//Calculate thrust and torque
Thrust = KT*sqr(n)*sqr(sqr(2*propRad));
Torque = KQ*sqr(n)*2*propRad*sqr(sqr(2*propRad));

totvFA=0;
totvFT=0;
ninside=0;
volinside=0;

//Loop through cell centres
forAll (mesh.C(), celli)
{
        cellC = mesh.C()[celli];        //Coordinate of current cell centre

        //Pojection of vector from prop centre to cell centre on propeller axis
           ↪ ;
        projPaxis = propOrient*(propOrient & (cellC-propOrigin));

        //If this cell centre is within the thickness of the propeller disk,
           ↪ proceed.
        if(mag(projPaxis)<=thickness/2)
        {

                //Pojection of vector from prop centre to cell centre on
                   ↪ propeller plane;
                projPplane[0]=(cellC[0]-propOrigin[0])*(sqr(propOrient[1])+sqr(
                   ↪ propOrient[2]))
                     -(cellC[1]-propOrigin[1])*propOrient[0]*propOrient[1]
                     -(cellC[2]-propOrigin[2])*propOrient[0]*propOrient[2];
                projPplane[1]=-(cellC[0]-propOrigin[0])*propOrient[0]*
                   ↪ propOrient[1]
                      +(cellC[1]-propOrigin[1])*(sqr(propOrient[0])+sqr(
                         ↪ propOrient[2]))
                      -(cellC[2]-propOrigin[2])*propOrient[1]*propOrient[2];

                projPplane[2]=-(cellC[0]-propOrigin[0])*propOrient[0]*
                   ↪ propOrient[2]
                      -(cellC[1]-propOrigin[1])*propOrient[1]*propOrient[2]
                      +(cellC[2]-propOrigin[2])*(sqr(propOrient[0])+sqr(
                         ↪ propOrient[1]));

                //Radius of current cell centre to propeller origin
                currRad = mag(projPplane);

                //IF this cell centre is within the radius of the propeller
                   ↪ disk, proceed.
                if(currRad<=propRad && currRad >= hubRad)
                {
                        //Mark cells inside disk and add their volume to total
                           ↪ volume calculation
                        cellsinside.append(celli);
                        cellradii.append(projPplane);
                        cellV = mesh.V()[celli];           //Volume of current
                           ↪ cell centre
                        volinside+=cellV;                  //Total volume

                        //Non-dimensional radius (0 at hub, 1 at tip)
                        radiusScaled = (currRad/propRad-hubRad/propRad)/(1.0-
                           ↪ hubRad/propRad);

                        //Calculate total force of the normalised distribution
                           ↪ in order to correct for KT and KQ later
                        totvFA += radiusScaled*Foam::sqrt(1.0-radiusScaled)*
                           ↪ cellV;
                        totvFT += radiusScaled*Foam::sqrt(1-radiusScaled)/(
                           ↪ radiusScaled*(1-hubRad)+hubRad)*cellV*currRad;

                        //Step up counter
                        ninside++;

                }
                else
                        {
                                volumeForce[celli]=vector::zero;
                        }
        }
        else
        {
                volumeForce[celli]=vector::zero;
        }
}

//If propeller cells are in different processors, make sure the sums are added.
reduce(ninside,maxOp<scalar>());
reduce(volinside,maxOp<scalar>());

//Distribute volume force over cells inside disk to achieve set thrust and
   ↪ torque
forAll(cellsinside,cellIi)
```

```
{
        currRad = mag(cellradii[cellIi]);


        //Non dimensional radius
        radiusScaled = (currRad/propRad-hubRad/propRad)/(1.0-hubRad /propRad);

        //Axial force distribution
        axF=radiusScaled*Foam::sqrt(1.0-radiusScaled);

        //Tangential force distribution
        tanF = radiusScaled*Foam::sqrt(1-radiusScaled)/(radiusScaled*(1-hubRad)
           ↪ +hubRad);

        //Tangential direction (orthogonal to radial direction)
        vector tanDir = propOrient ^ cellradii[cellIi];
        //Make tangential vector unit length
        tanDir = tanDir/mag(tanDir);

        //Finally, calculate the volume force distribution and make sure the
           ↪ Thrust and Torque are matched
        volumeForce[cellsinside[cellIi]] = propOrient*Thrust*axF/(totvFA +
           ↪ VSMALL) + tanDir*Torque*tanF/(totvFT+VSMALL);

}


Info<<"Found "<<ninside<<" cells in propeller disk"<<endl;
Info<<"Cell volume / disk volume = "<<100*volinside/(constant::mathematical::pi
   ↪ *thickness*(sqr(propRad)-sqr(hubRad))<<" %"<<endl;


Info<<"U0 : "<<U0<<" ,  J : "<<J<<endl;
Info<<"KT : "<<KT<<" ,  KQ : "<<KQ<<endl;
```

7 Compiling the new solver

After following the steps in the previous sections, to compile the solver run the following command in the *my_simpleFoam*-directory

```
wmake
```

After successful compilation, *my_simpleFoam* is now available as a solver to use in any OpenFOAM case. Bear in mind that the case must contain the *propellerDict* dictionary, in the form shown in Section 5 and the F_v boundary conditions, in the form shown in Section 3 for the solver to work.

8 Testing

This section shows a very simple usage example for the *my_simpleFoam* solver that was created in the previous sections. The propeller is modeled inside a cylindrical domain with a uniform inflow speed of $U_\infty = 20m/s$. A shaft extends from the inlet to the propeller position to mimic the setup of a cavitation tunnel. The dimensions of the domain are shown in Fig. 1. The propeller radius is $r_0 = 1m$ and the rotation rate is $n = 20/s$ giving an advance ratio of $J = 0.5$. The OpenFOAM case used here is attached to this document.

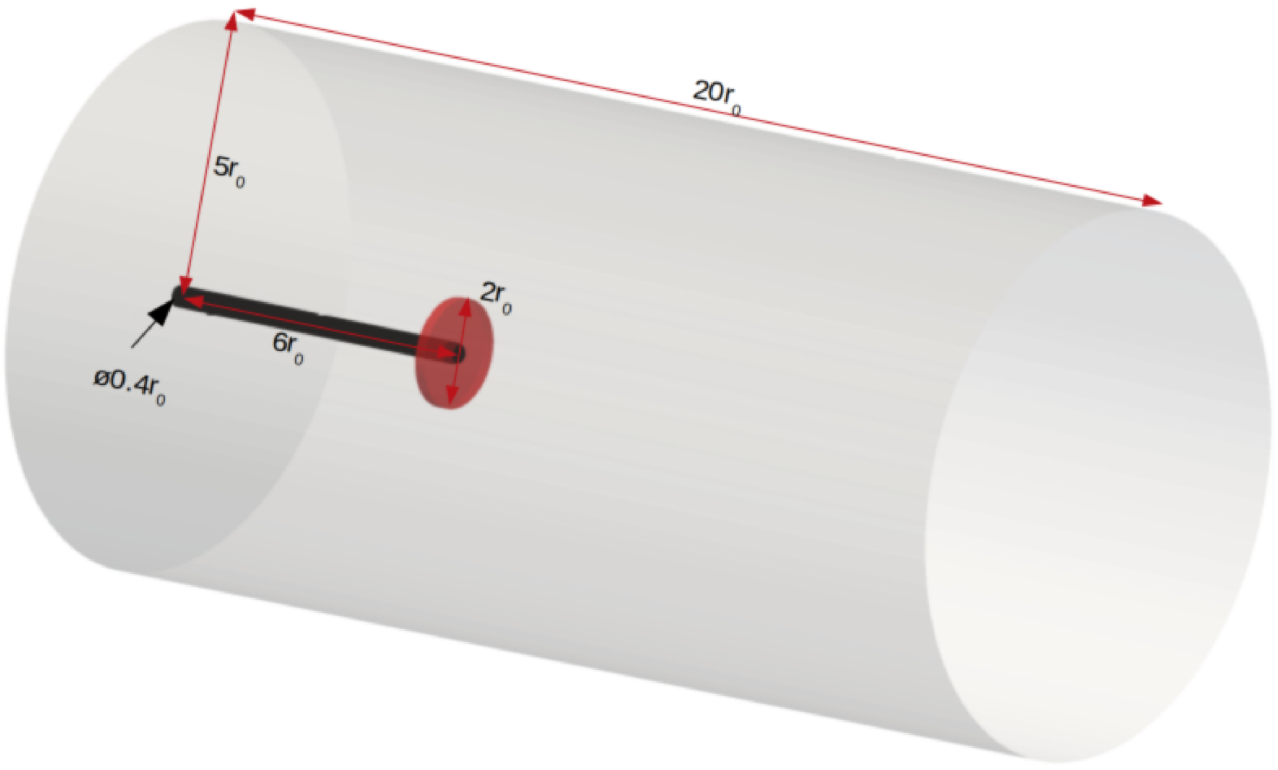

Figure 1: Computational domain: a circular cylinder with a shaft protruding downstream from the inlet. The area where the body force is active (the location of the propeller) is highlighted but is purely virtual. No physical propeller or discretization thereof exist.

This example serves only to illustrate that the solver serves its purpose of accelerating the flow and does not go into detail regarding the flow-field accuracy with regards to the target propeller. For more advanced propeller models, such an analysis would be necessary to determine their fitness for purpose. Because the inflow velocity in this solver is probed at a single point upstream, it is unsuitable for studying propeller/hull interaction. It would however be possible to use it for limited propeller/rudder interaction studies by placing a rudder downstream.

The pressure distribution on the tunnel center plane is shown in Fig. 2. The pressure jump stemming from the propeller blades can clearly be seen. Note that the pressure jump is not explicitly modeled or enforced but is a result of the pressure-velocity coupling and the applied momentum source (F_v, see Eq. (1)). Three-dimensional streamlines are also shown to illustrate how the flow is rotated at the propeller plane.

The axial velocity on the tunnel center plane is shown in Fig. 3, with a detailed view near the propeller in Fig. 4. Three-dimensional streamlines are shown in these figures to illustrate how the flow is rotated.

References

Goldstein, S. (1929). On the vortex theory of screw propellers. *Proc. R. Soc. Lond.*, 123.

Windén, B. (2021a). An open-source framework for ship performance cfd. In *Proceedings of the 26th SNAME Offshore Symposium.*

Windén, B. (2021b). Predicting the powering performance of different vessel types using an open-source cfd propulsion framework. In *Proceedings of the SNAME Maritime Convention 2021.*

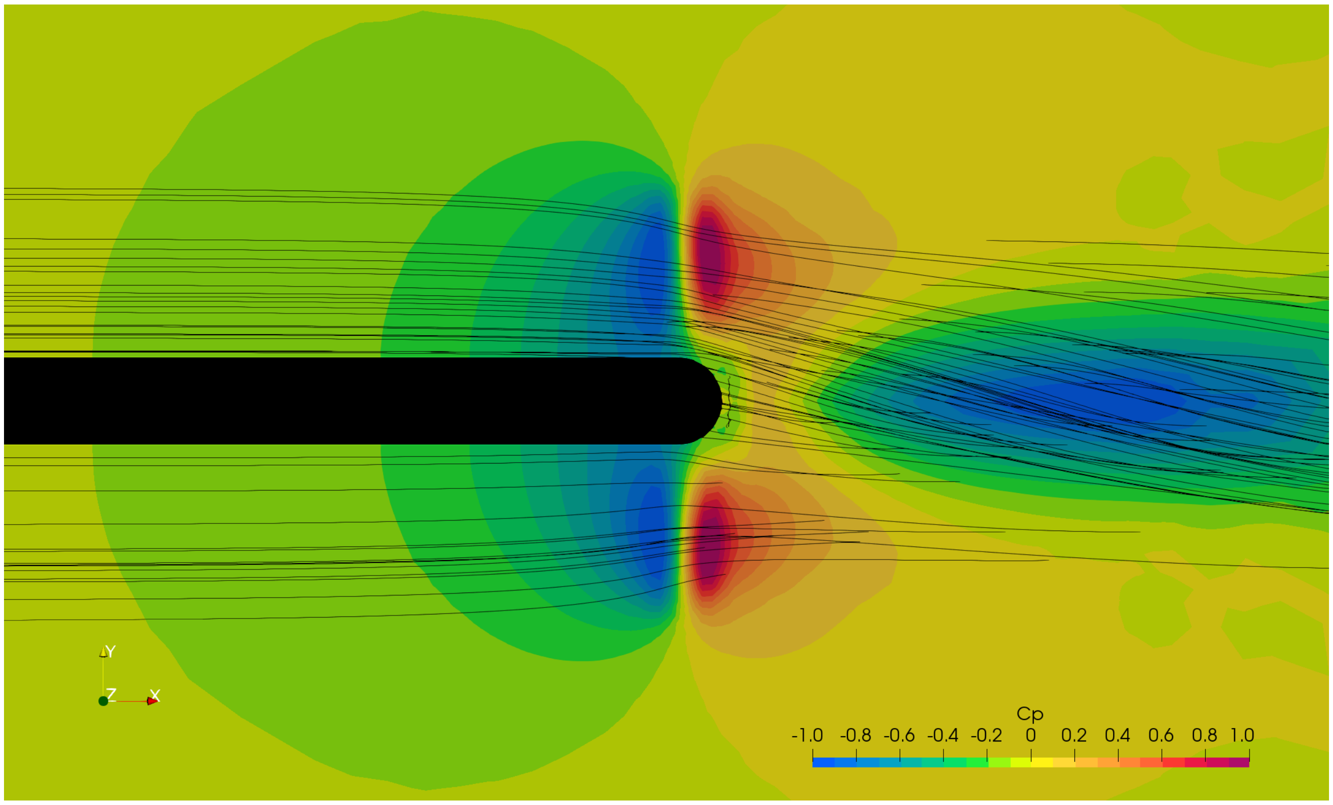

Figure 2: Pressure coefficient C_p on the center plane of the tunnel. Streamlines (in 3D) passing through the propeller disk are show in order to illustrate how the flow is rotated.

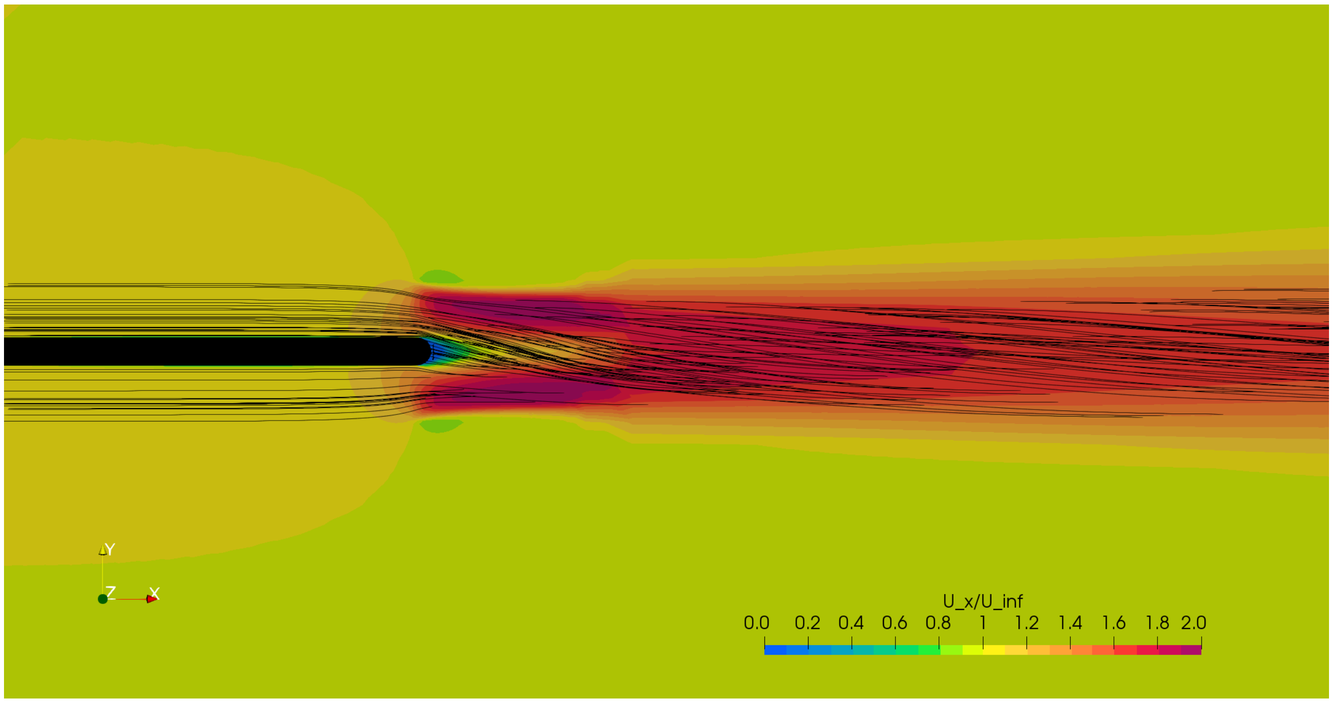

Figure 3: Relative axial velocity U_x/U_∞ on the center plane of the tunnel. Streamlines (in 3D) passing through the propeller disk are show in order to illustrate how the flow is rotated.

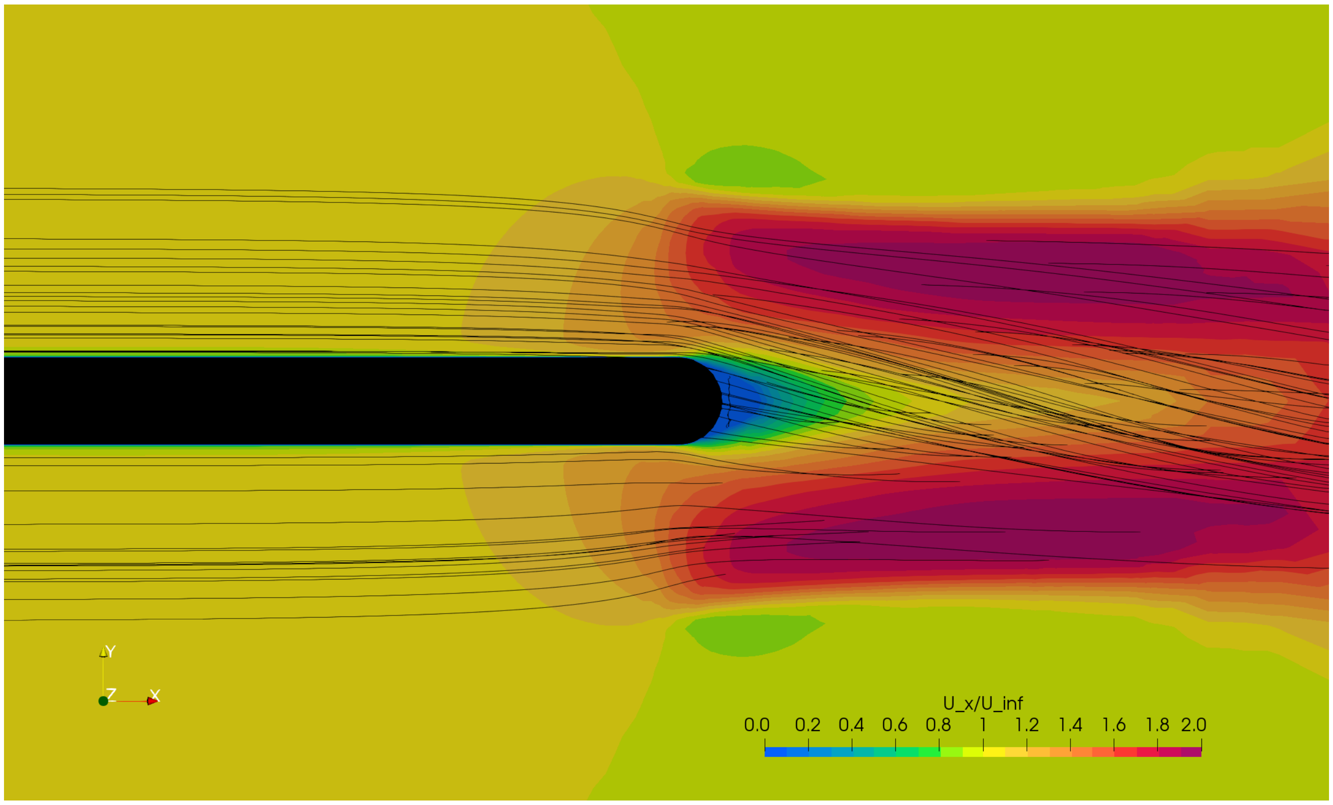

Figure 4: Detail (near the propeller) of relative axial velocity U_x/U_∞ on the center plane of the tunnel. Streamlines (in 3D) passing through the propeller disk are show in order to illustrate how the flow is rotated.