CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Implementation for lifting line propeller representation

Developed for OpenFOAM-2.1.x

Author:
Florian Vesting

Peer reviewed by:
Mostafa Payandeh
Jelena Andric

November 5, 2012

Chapter 1

Background

1.1 Introduction

Propeller design is a highly complex procedure involving many influencing factors. Common practice is to develop a preliminary design concept and improve this by finding the best compromise between the objectives and constraints. Starting from scratch the initial geometry needs to be developed, by formulating the requirements on the propeller blade geometry through the propeller load distribution in radial direction and assuming that the radial distribution of circulation is related to the actual blade geometry. This is usually done by applying lifting line theory.

In lifting line theory the actual blade geometry is replaced by span-wise panels of constant line circulation, which generates lift when it experiences an inflow. The computational time to solve for the integrated blade lift and drag force is insignificant compared to a the case when the detailed flow simulation around the blade needs to be computed by a high-fidelity computational method like RANS. Thus the lifting line method provides a reasonable alternative to compute general propeller characteristics and offers a widely excepted method to simulate self-propelled ship resistance in hybrid-RANS computations.

1.2 Lifting Line Theory

The lifting line method is a mathematical rather plain approach to compute the lift of a wing. It is based on the classical lifting line theory [6], which is adapted to the marine propeller problem in Lerbs analysis method for *moderately loaded propeller* [4]. The method assumes the propeller blade sections to be replaced by a single line vortex that varies in strength from section to section. The line, about which the vortices act, is a continuous in radial direction. Figure 1.1 shows the discretisation of the propeller geometry by a lifting line. This figure shows also the free vortices shed from the each bound (lifting) vortex along the lifting line, to satisfy the Helmholtz's theorem of the principles of inviscid vortex behaviour.

The performance of the propeller is strongly related to its inflow and needs be taken into account. Generally this is done at an early design stage through the mean circumferential wake distribution. Methods were developed to determine the optimal circulation distribution for a given wake distribution. In this context optimum distribution refers to circulation that is minimized regarding thrust creation. Goldstein presented in [3] the first work on optimal circulation distribution on screw propellers. Lerbs developed an efficient algorithm to calculate the induced velocities of the lifting lines of a propeller and criteria on induced velocities for the optimum propellers in non-uniform inflow in [4]. These criteria can be used in turn to obtain the optimum circulation for a given inflow. Wrench [7] developed the proposed method by Lerbs further, to obtain more accurate formulas for the induction factors. With the obtained optimal circulation, the geometric parameters can be elaborated afterwards.

The implemented procedure follow the implementation of openProp, an open source code that

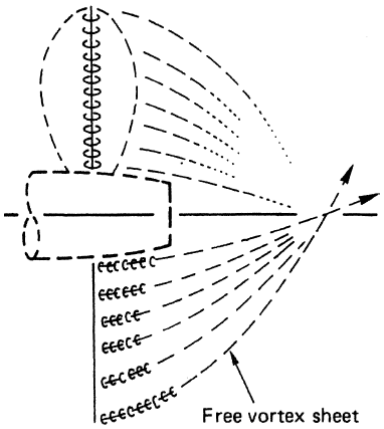

Figure 1.1: Hydrodynamic model of a propeller by lifting line theory, figure taken from [1]

can be used for the design, analysis, and fabrication of optimized propellers and horizontal-axis turbines. The method is valid for *moderately loaded propellers* and employs formulas developed in [7]. The induced velocities are computed using the free vortices shed at each discrete station along the blade. Figure 1.2 shows the velocities and forces on a 2-dimensional (2D) blade section [5]. $\mathbf{e_a}$ is the axial direction of the propeller and $\mathbf{e_t}$ is the tangential direction. $\mathbf{V}^*$ is the total resultant inflow velocity at the certain blade section, as a result from the axial inflow $\mathbf{V_a}$, the tangential inflow $\omega r + \mathbf{V_t}$, and the induced axial and tangential velocities $\mathbf{u_a^*}$ and $\mathbf{u_t^*}$. The inviscid lift force generated at the 2D section i is eventually computed by according to equation 1.1. The viscous drag force is calculated according to equation 1.2 and aligned with the total inflow velocity $\mathbf{V}^*$, with the local profile drag coefficient C_D and the local section length c.

$$\mathbf{F_i} = \rho \mathbf{V}^* \times (\Gamma \mathbf{e_r}) \tag{1.1}$$

$$\mathbf{F_v} = \frac{1}{2}\rho (V^*)^2 C_D c \tag{1.2}$$

The induced velocities are computed on the vortex lattice formulation. The blade is partitioned

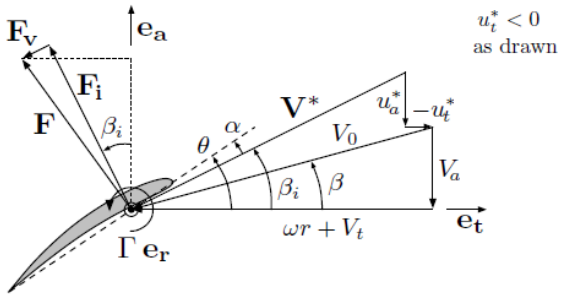

Figure 1.2: Propeller force diagram, figure taken from [5]

into M panel sections, with the $i-th$ panel having a horseshoe vortex filament with the circulation $\Gamma(i)$, consisting of the free vortices shed from the panel corner points ($r_v(i)$ and $r_v(i+1)$) and the vortex segment along the lifting line. The induced velocities are calculated on the control points on the lifting line at $r_c(m)(m = 1, ..., M)$, by summing the induced velocities of each horseshoe vortex.

$$u_a^*(m) = \sum_{i=1}^{M} \Gamma(i) \overline{u}_a^*(m, i) \tag{1.3}$$

$$u_t^*(m) = \sum_{i=1}^{M} \Gamma(i)\overline{u}_t^*(m,i) \tag{1.4}$$

$$\overline{u}_a^*(m,i) = \overline{u}_a(m,i+1) - \overline{u}_a(m,i) \tag{1.5}$$

$$\overline{u}_t^*(m,i) = \overline{u}_t(m,i+1) - \overline{u}_t(m,i) \tag{1.6}$$

With $\overline{u}_a^*(m,i)$ and $\overline{u}_t^*(m,i)$ being the axial and tangential velocities induced at the control point $r_c(m)$ by the surrounding panels. These are computed using the formulas by Wrench [7], according to 1.5 and 1.6. Since the lifting line itself does not contribute to the induced velocity, $\overline{u}_a(m,i)$ and $\overline{u}_t(m,i)$ are computed as follows:

For $r_c(m) < r_v(m)$:

$$\overline{u}_a(m,i) = \frac{Z}{4\pi r_c}(y - 2Zyy_0F_1) \tag{1.7}$$

$$\overline{u}_t(m,i) = \frac{Z^2}{y_0F_1} \tag{1.8}$$

For $r_c(m) > r_v(m)$:

$$\overline{u}_a(m,i) = -\frac{Z^2}{2\pi r_c}(yy_0F_2) \tag{1.9}$$

$$\overline{u}_t(m,i) = \frac{Z}{4\pi r_c}(1 + 2Zy_0F_2) \tag{1.10}$$

where

$$F_1 \approx \frac{-1}{2Zy_0}\left(\frac{1+y_0^2}{1+y^2}\right)^{\frac{1}{4}}\left\{\frac{U}{1-U} + \frac{1}{24Z}\left[\frac{9y_0^2+2}{(1+y_0^2)^{1.5}} + \frac{3y^2-2}{(1+y^2)^{1.5}}\right] ln\left|1 + \frac{U}{1-U}\right|\right\} \tag{1.11}$$

$$F_2 \approx \frac{1}{2Zy_0}\left(\frac{1+y_0^2}{1+y^2}\right)^{\frac{1}{4}}\left\{\frac{U}{U-1} + \frac{1}{24Z}\left[\frac{9y_0^2+2}{(1+y_0^2)^{1.5}} + \frac{3y^2-2}{(1+y^2)^{1.5}}\right] ln\left|1 + \frac{U}{U-1}\right|\right\} \tag{1.12}$$

$$U = \left(\frac{y_0\left(\sqrt{1+y^2}-1\right)}{y\left(\sqrt{1+y_0^2}-1\right)}\exp\left(\sqrt{1+y^2} - \sqrt{1+y_0^2}\right)\right) \tag{1.13}$$

$$y = \frac{r_c}{r_v tan\beta_w} \tag{1.14}$$

$$y_0 = \frac{1}{tan\beta_w} \tag{1.15}$$

In order to align the free trailing vortices with the local flow at the blade we set $\beta_w = \beta_i$. This holds for all moderately-loaded propellers. To account for the propeller hub, an image trailing vortex filament can be modelled with equal strength but opposite direction. This is however not yet included in this project.

1.3 Windturbine Class

The foundation for the lifting line implementation is given by the class *horizontalWindTurbineArray*. Matthew J. Churchfield [2] presented an Actuator Line Turbine Model to resolve the turbine blade geometry in a hybrid high-Re LES computations. In this method the turbine blades are discretized into spanwise sections of constant airfoil properties. The lift and drag forces are calculated by the class *horizontalWindTurbineArray* by looking up pre-specified turbine and section foil properties in input files. Once the forces at a specific radial position, at a certain time step are calculated with

respect to the local inflow, the turbine forces are introduced at the momentum equation of the solver code.

The *horizontalWindTurbineArray* class already provides many functions for the analysis of wind turbines that are equally useful for the simulation of ship propellers. First of all it is the turbine performance updated every time step, which makes it possible to take unsteady inflow into account. Furthermore it is possible to evaluate various number of turbines in one domain during one simulation, in which each turbine can be of different geometric characteristics. This is of particular interest in propeller design, when designing contra-rotating propellers. With the function for yaw control a podded propulsion system simulates manoeuvring of a whole ship. The code is already capable to run in parallel and extract the velocity field at the blade position even if the blade exceeds the decomposed grid boundaries.

Chapter 2

Implementation

2.1 Lifting Line Class

The new *openPropLiftingLine* class is based on the *horizontalWindTurbineArray* class. To keep the focus on the propeller performance analysis, all member function that are not needed for the propeller, were removed from the original code. Only functions that will be of particular interest in future work are still included in the code, e.g. functions to write the output files. The implemented version predicts the forces from a propeller blade for a given circulation distribution and introduces the volume forces to the underlying volume grid. However, the purposed implementation is not the complete. An optimisation of the blade circulation with respect to the inflow is omitted.

2.1.1 Requirements

The user needs to provide several basic inputs to the code, to characterize the propeller. The information are specified within at least two files. The file *propellerArrayProperties* specifies one or several propellers, where their type, initial state and the general information of the lifting line method are provided. The inputs in this file are summarized in table 2.1. The second file the user needs to provide is the propeller properties file. The name is given by *propellerType* in the *propellerArrayProperties* for each propeller. Within this file the detailed geometric propeller characteristics are given. The desired inputs are given in table 2.2.

Input	General meaning
propellerType	Type of the propeller to find its characteristics-file
baseLocation	Location of the boss on the ships baseline
numBladePoints	Number of control points along the lifting line
pointDistType	Point distribution along the lifting line (not implemented)
epsilon	Gaussian width parameter for the projection of the point force onto the grid
smearRadius	Radius beyond which Gaussian has no effect
sphereRadiusScalar	Size of the sphere around propeller for grid point search
tipRootLossCorrType	
rotationDir	Rotation direction viewed in upstream direction
Azimuth	Initial position of the blades
RotSpeed	Propeller speed in RPM
NacYaw	Propeller direction towards the inflow (270.0 = West)

Table 2.1: Inputs in *propellerArrayProperties*

Figure 2.1 shows schematically two possible general layouts for the propeller lifting line approach. On the left side for a conventional single screw propulsion set-up behind the ship. And on the right

Input	General meaning
NumBl	Number of blades
TipRad	Blade tip radius
HubRad	Hub radius
Vs	Ship speed for normalization
CTPDES	Desired thrust coefficient for initial estimations
Overhang	Distance between boss and propeller plane
Baseline2Shft	Vertical distance between shaft and baseline
ShftTilt	Shaft inclination in deg.
Rake	Blade rake at the tip in deg.
YawRate	Rotational speed for yawing (podded propulsion, not implemented)
SpeedControllerType	Speed control for rotation
YawControllerType	Speed control for yawing
BladeData	Blade properties along the radius (chord, Γ, drag coefficient)

Table 2.2: Inputs in the propeller specific file

side, for a podded propulsion unit with two propellers. The compass directions (N, E, S, W) discribe the cardinal directions for the initial state of the propeller, adopted from Churchfield.

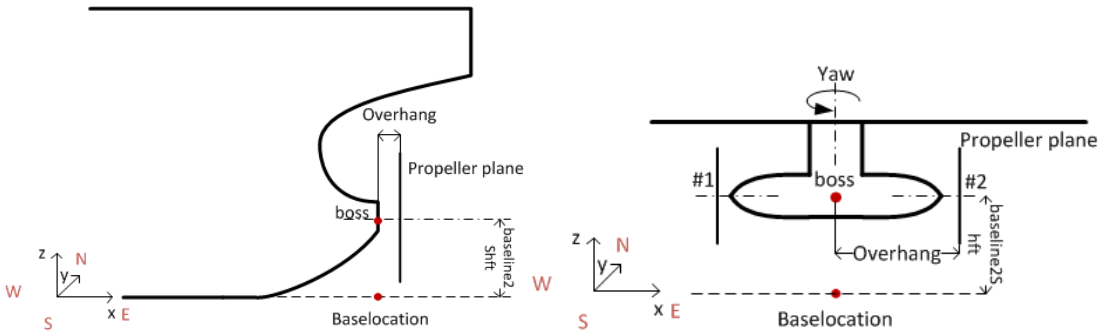

Figure 2.1: Layout of the propeller lifting line approach, for a conventional single screw hull (left), or a podded propulsion unit (right)

2.1.2 Assumptions and simplification

The intention in this course project is to implement the functionality of the OpenPROP matlab toolbox into openFOAM. This includes the force integration of the blade, based on the circulation along the blade, as well as the optimisation of the circulation and an initial blade design. The optimisation problem is to find M circulations of the vortex panels that produce the least torque for a specified thrust constraint. In OpenPROP this is done by employing the method of the Lagrange multiplier to find the optimum of the auxiliary function $H = Q + \lambda(T - T_s)$, where Q is the torque, T the total thrust generated by the propeller and T_s is the required thrust. It results in a non-linear system of a M+1 equations and M+1 unknowns, which are solved iteratively. However, the first step in this project covers only the force calculation for a given circulation, as outlined in section 1.2. Furthermore, the β_i for the Wrench induction factors is computed at the control points and later interpolated to the vertices by a piecewise cubic hermite interpolation in the original matlab code. For the time being it is assumed to be sufficient to interpolate β_i with a simple piecewise linear interpolation function and set the β_i at the outermost vertices equal to the values at the first and last control points respectively.

2.1.3 General class structure

In general the lift and drag forces acting on a single 2D section as shown in figure 1.2 are introduced as a volume source and added to the momentum equation in the solver code. When the propeller rotates in the wake of a ship, it experiences different inflow velocities and different angles of attack. Hence, the generated blade force depends also on the current inflow at the position of the blade. Thus a transient solver is most appropriate to analyse the propeller characteristics. We will use the class with the standard transient openFOAM solver *pisoFoam* for incompressible flow. With this solver various turbulence models can be applied to supply an accurate inflow behind a ship. This section will provide an overview of the class and the general structure of the model, before we take a closer look at each of the functions in the new class. The class consists basically of the header file *openPropLiftingLine.H*, containing the declaration of the member functions, variables, constructor and destructor and the source code *openPropLiftingLine.C*. It follows the general features of the new class:

- Create an object of the *openPropLiftingLine* class in the solvers header file
  - Initialize *const* pointers to the *runTime, mesh* and *velocity field*
  - Initialize a new vector field for the body forces
  - Read in the dictionary *propellerArrayProperties* which is the uppermost level dictionary that describes where the propellers are, what kind they are and their initial state
  - Read in dictionaries for each of the propellers specified in *propellerArrayProperties*.
  - Determine the cells that can possibly be influenced by the propeller forces.
  - Discretise the lifting line in M panels and keep the location of each control and each vertex point
  - Initialize the dynamic lists of scalar or vectors for all forces and flow velocities according to the number of control points
  - Rotate the propeller to its initial yaw and azimuth position
  - For the case of multiple processors, find out which control point is controlled by which processors
  - Compute the inflow for each control point and align it with the local blade coordinate system
  - Iterate the induced velocities, by starting from an estimation of $\mathbf{u}^*$ to find the induced pitch angle β_i for the computation of the horseshoe influence, equation 1.7 and 1.8
  - Compute the blade forces and transform them to the global coordinate system
  - Compute the body forces
- Call the class *void update()* function inside the *runTime.loop()*

2.1.4 openPropLiftingLine.C

In this section we take a closer look at the function in the *openPropLiftingLine* class. The general outline of the class structure is already given in section 2.1.3. The class contains various functions, which are partly adopted from the initial class *horizontalWindTurbineArray* by Churchfield. This section, however, will be restricted to functions which are newly included within the course project and function that are essential for the understanding of the code.

- First of all we need to get the current time step `dt` which is later used in the *computeRotSpeed()* function to determine the current position of the propeller. This is done at the beginning of the class and in the *update()* function.

Source Code: openPropLiftingLine.C

```
56      // Set the time step size.
57      dt(runTime_.deltaT().value()),

649         forAll(rotSpeed, i)
650         {
651         int j = propellerTypeID[i];
652         if (SpeedControllerType[j] == "none")
653         {
654             // Do nothing.
655             deltaAzimuth[i] = rotSpeed[i] * dt;
656         }
657         else if (SpeedControllerType[j] == "simple")
658         {
659             // Placeholder for when this is implemented.
660     }
661     }
662 }
```

- To communicate data, we create an object of the standard *IOobject* class at the very beginning of our class. This provides standardized input/output support, to simplify the communication between the solver and the data. The following part of the code was already provided in the *horizontalWindTurbineArray* class for the body forces and dictionaries.

Source Code: openPropLiftingLine.C

```
62      // Initialize the body force.
63      bodyForce
64      (
65          IOobject
66          (
67              "bodyForce",
68              time,
69              mesh_,
70              IOobject::NO_READ,
71              IOobject::AUTO_WRITE
72          ),
73          mesh_,
74          dimensionedVector("bodyForce",dimForce/dimVolume/dimDensity,vector::zero)
75      )
76
77  {
78      // Define dictionary that defines the propeller array.
79      IOdictionary propellerArrayProperties
80      (
81          IOobject
82          (
83              "propellerArrayProperties",
84              runTime_.constant(),
85              mesh_,
86              IOobject::MUST_READ,
87              IOobject::NO_WRITE
88          )
89      );
```

- The class uses dynamic list for the variable to be able to store data and loop over various numbers of propellers, number of blades on each propeller and their discretisation. For instance baseLocation of the propellers is stored in a list of vectors for each propeller. The declaration in the header file and the use in the source code are given below as an example how data are read from the dictionary.

Source Code: openPropLiftingLine.H

```
112             //- List of locations of bases of propellers in array relative to
                    origin (m).
113             DynamicList<vector> baseLocation;
```

Source Code: openPropLiftingLine.C

```
96      propellerName = propellerArrayProperties.toc();
97
98      numPropellers = propellerName.size();
99
100     forAll(propellerName,i)
101     {
102         propellerType.append(word(propellerArrayProperties.subDict(propellerName[
                i]).lookup("propellerType")));
103         baseLocation.append(vector(propellerArrayProperties.subDict(propellerName
                [i]).lookup("baseLocation")));
```

- The next important step is to identify the grid cells whose velocities can possibly be influenced by the force introduced from each propeller. This is done by defining a sphere of cell IDs around a propeller, which will be saved in the memory. This is done in the class once, when an instance is created from the solver, by going through all cells in the mesh and evaluating the distance to the propeller root.

Source Code: openPropLiftingLine.C

```
289         forAll(U_.mesh().cells(),cellI)
290         {
291             if (mag(U_.mesh().C()[cellI] - boss[i]) <= sphereRadius)
292             {
293             sphereCellsI.append(cellI);
294             }
295         }
296         sphereCells.append(sphereCellsI);
297         sphereCellsI.clear();
298         }
```

- In this context also specify normalized vectors for the global coordinate system, which is done within the loop to discretise over all propellers and all the blades. The *uvShaft* and *uvZ* are determined by the input coordinate of the propeller, while *uvY* is created perpendicular to the first two vectors. All vectors are normalized with their magnitude.

Source Code: openPropLiftingLine.C

```
311         // Define which way the shaft points to distinguish between
312         // upwind and downwind propellers.
313         uvShaftDir.append(OverHang[j]/mag(OverHang[j]));
314
315         // Define the vector along the shaft pointing in the
316         // direction of the wind.
317         uvShaft.append(rotorApex[i] - boss[i]);
318         uvShaft[i] = (uvShaft[i]/mag(uvShaft[i])) * -uvShaftDir[i];
319
320         // Define the vector aligned with the vertical coordinate pointing from
321         // the ground to the boss.
322         uvZ.append(boss[i] - baseLocation[i]);
323         uvZ[i] = uvZ[i]/mag(uvZ[i]);
324
325         uvY.append( uvShaft[i] ^ uvZ[i]);
326         uvY.append( uvY[i] / mag(uvY[i]));
```

- The length of each propeller section and the panel length `db` is calculated within the same loop. At the moment only a uniform distribution of control points is possible, e.g. `db` is the same over all radii.

Source Code: openPropLiftingLine.C

```
327         // Calculate the length of each propeller section.
328         db.append(DynamicList<scalar>(0));
329         if(pointDistType[i] == "uniform")
330         {
331             scalar liftingLineLength = (TipRad[j]-HubRad[j])/numBladePoints[i];
332             for(int m = 0; m < numBladePoints[i]; m++)
333             {
334             db[i].append(liftingLineLength);
335             }
336         }
```

- Once the spacing is set, the vectors to each control and vortex point are computed at the blade pointing upward orientation. The starting position of each blade and the position at each time step is calculated by the function *rotatePoint()*, which returns a vector of the new coordinates based on the rotation angle. Now, first the *arrays* for the blade points and vortex points are initialized with zero vectors. In fact, the vectors of the points are listed. There is for each blade a list of vectors according to the the number of control points. This lists are again listed for each propeller according to the number of blades. Then a loop is started for each blade of that particular propeller. Here the *root* vector is computed at first, which is the starting point for the control points. The vertex points start from the propeller vortex location. Within a loop over all blade points, the x, y and z coordinates are computed by gradually increasing the distance `dist` from the starting point. The last vertex position needs to be added outside the loop at the tip radius `tipRad` .

Source Code: openPropLiftingLine.C

```
345         bladePoints.append(List<List<vector> >(NumBl[j], List<vector>(
                numBladePoints[i],vector::zero)));
346         bladeVortexPoints.append(List<List<vector> >(NumBl[j], List<vector>(
                numBladePoints[i]+1,vector::zero)));
347         bladeRadius.append(List<List<scalar> >(NumBl[j], List<scalar>(
                numBladePoints[i],0.0)));
348         bladeVortexRadius.append(List<List<scalar> >(NumBl[j], List<scalar>(
                numBladePoints[i]+1,0.0)));
349
350         for(int k = 0; k < NumBl[j]; k++)
351         {
352             int tip = 0;
353             vector root = rotorApex[i];
354             vector apex = rotorApex[i];
355             scalar beta = Rake[j][k] - ShftTilt[j];
356             root.x() = root.x() + HubRad[j]*Foam::sin(beta);
357             root.z() = root.z() + HubRad[j]*Foam::cos(beta);
358             scalar dist = HubRad[j];
359             for(int m = 0; m < numBladePoints[i]; m++)
360             {
361                 bladeVortexPoints[i][k][m].x() = apex.x() + dist*Foam::sin(beta);
362                 bladeVortexPoints[i][k][m].y() = apex.y();
363                 bladeVortexPoints[i][k][m].z() = apex.z() + dist*Foam::cos(beta);
364                 bladeVortexRadius[i][k][m] = dist;
365                 dist = dist + 0.5*db[i][k];
366                 bladePoints[i][k][m].x() = root.x() + dist*Foam::sin(beta);
367                 bladePoints[i][k][m].y() = root.y();
368                 bladePoints[i][k][m].z() = root.z() + dist*Foam::cos(beta);
369                 bladeRadius[i][k][m] = dist;
```

```
370                totBladePoints++;
371                dist = dist + 0.5*db[i][k];
372                tip = m+1;
373            }
374                    // include outermost vortex location
375                    bladeVortexPoints[i][k][tip].x() = apex.x() + dist*Foam::sin(
                           beta);
376            bladeVortexPoints[i][k][tip].y() = apex.y();
377            bladeVortexPoints[i][k][tip].z() = apex.z() + dist*Foam::cos(beta);
378            bladeVortexRadius[i][k][tip] = dist;
379            // Apply rotation to get blades, other than blade 1, in the right
380            // place.
381            if (k > 0)
382            {
383            for(int m = 0; m < numBladePoints[i]; m++)
384            {
385                bladePoints[i][k][m] = rotatePoint(bladePoints[i][j][m],
                       rotorApex[i], uvShaft[i], (360.0/NumBl[j])*k*degRad);
386                bladeVortexPoints[i][k][m] = rotatePoint(bladeVortexPoints[i][j][
                       m], rotorApex[i], uvShaft[i], (360.0/NumBl[j])*k*degRad);
387                        tip = m+1;
388            }
389                    // include outermost vortex location
390                    bladeVortexPoints[i][k][tip] = rotatePoint(
                           bladeVortexPoints[i][j][tip], rotorApex[i], uvShaft[i
                           ], (360.0/NumBl[j])*k*degRad);
391
392            }
393        }
```

- When knowing the number of control and vertex points, the dynamic lists for all needed variables can be initialized as e.g. $\mathbf{V}^*$ or its magnitude V^*, see below.

Source Code: openPropLiftingLine.C

```
407                // Define the total inflow velocity array and set it to zero.
408                VSTAR.append(List<List<vector> >(NumBl[j],List<vector>(
                       numBladePoints[i],vector::zero)));
409
410                //- Define the magnitude array of the total inflow velocity
411                VSTARmag.append(List<List<scalar> >(NumBl[j],List<scalar>(
                       numBladePoints[i],0.0)));
```

- The so far presented code is executed when instantiating the class. However, the now following function are also called from the public member function *void update()*, which is called every time step from the solver. The propeller is turned to the its initial yaw and azimuth angle by the function *yawNacelle()* and *rotateBlades()*. Next, the *findControlProcNo()* function is called to Find out which processors control each control point. When knowing which processor controls which control point, the inflow velocities are determined at the control point locations in the *void computeWindVectors()* function. The local inflow to a control point is then aligned with blade orientation. The code to find the blade orientation over each propeller (i) at each blade (j) is given below. In case of a clockwise rotating propeller, the local blade vector $\mathbf{e_r}$ points from the root along the blade to the tip; this is `bladeAlignedVectors[i][j][2]`. The local blade vector in tangential direction $\mathbf{e_t}$ (`bladeAlignedVectors[i][j][1]`) is perpendicular to the shaft and $\mathbf{e_r}$ and points tangential in rotation direction. Further down are the inflow vectors (`windVectors[i][j][k]` aligned in each direction as the scalar component of the local blade coordinates; point by point for the current propeller, on the current blade. The inflow is directly normed with the ships velocity and the rotational speed is added to the y-component.

Source Code: openPropLiftingLine.C

```
722         if (rotationDir[i] == "cw")
723         {
724                 bladeAlignedVectors[i][j][2] =   bladePoints[i][j][0] - rotorApex
                        [i];
725                 bladeAlignedVectors[i][j][2] =   bladeAlignedVectors[i][j][2]/mag
                        (bladeAlignedVectors[i][j][2]);
726         }
727         else if (rotationDir[i] == "ccw")
728         {
729                 bladeAlignedVectors[i][j][2] = -(bladePoints[i][j][0] - rotorApex
                        [i]);
730                 bladeAlignedVectors[i][j][2] =   bladeAlignedVectors[i][j][2]/mag
                        (bladeAlignedVectors[i][j][2]);
731         }
732
733             bladeAlignedVectors[i][j][1] = bladeAlignedVectors[i][j][2]^uvShaft[i
                    ];
734             bladeAlignedVectors[i][j][1] = bladeAlignedVectors[i][j][1]/mag(
                    bladeAlignedVectors[i][j][1]);
735
736             bladeAlignedVectors[i][j][0] = bladeAlignedVectors[i][j][1]^
                    bladeAlignedVectors[i][j][2];
737             bladeAlignedVectors[i][j][0] = bladeAlignedVectors[i][j][0]/mag(
                    bladeAlignedVectors[i][j][0]);
738
739             // Proceed point by point.
740             forAll(windVectors[i][j], k)
741             {
742                 // Zero the wind vector.
743                 windVectors[i][j][k] = vector::zero;
744
745                 // Now put the velocity in that cell into blade-oriented
                        coordinates.
746                 windVectors[i][j][k].x() = (bladeAlignedVectors[i][j][0] &
                        windVectorsLocal[controlProcNo[i][j][k]][iter]) / VS[i];
747                 windVectors[i][j][k].y() = (bladeAlignedVectors[i][j][1] &
                        windVectorsLocal[controlProcNo[i][j][k]][iter]) / VS[i] + (
                        rotSpeed[i] * bladeRadius[i][j][k] * cos(Rake[n][j]));
748                 windVectors[i][j][k].z() = (bladeAlignedVectors[i][j][2] &
                        windVectorsLocal[controlProcNo[i][j][k]][iter]) / VS[i];
749                 iter++;
750             }
```

- Now we need to correct the inflow velocities with propeller induced velocities. This is done in the *void iterateInducedVelocities()* function. At the moment this function iterates for a fixed number of iterations to converge the induced velocities. Tests have shown that the selected number iterations is sufficient. One could instead easily evaluate the maximum difference between the velocity magnitudes. However, when the class will be extended to eventually optimise the circulation distribution, the induced velocities will be updated within the circulation iteration. The following source code shows the procedure to evaluate the induced velocities.

Source Code: openPropLiftingLine.C

```
1129 //- Iterate induced velocities
1130 void openPropLiftingLine::iterateInducedVelocities()
1131 {
1132     int x = 0;
1133     estimateUSTAR();
1134     findTanBetaI();
1135     while ( x < 5)
1136     {
1137         findTanBetaI();
```

```
1138         // compute horseshoe influence
1139         horseShoe();
1140         // compute VSTAR derivative NOT IMPLEMENTED
1141
1142         // start circulation distribution optimisation
1143         // set up simultaneous equations for G and LM
1144         // and solve linear system of equation -NOT IMPLEMENTED
1145
1146         // compute induced velocities
1147         inducedVelocity();
1148
1149         // compute VSTAR magnitude
1150         computeVSTARmag();
1151
1152         findTanBetaI();
1153         x++;
1154     }
1155 }
```

- First we have the *void estimateUSTAR()* function. This gives the first idea of the induced velocity in axial direction, to get the induced pitch angle β_i. The actual calculation of u_a^* follows equation 1.3, but the initial estimation is based on actuator disc theory, where $CTPDES$ is the use specified desired thrust coefficient (see table 2.2) :

$$u_a^* = \frac{1}{2}\left(\sqrt{1 + CTPDES} - 1\right) \tag{2.1}$$

Source Code: openPropLiftingLine.C

```
1034 //- Estimate the initial induced velocity
1035 void openPropLiftingLine::estimateUSTAR()
1036 {
1037     // for all propellers
1038     forAll(USTAR, i)
1039     {
1040         // for all blades
1041         forAll(USTAR[i], j)
1042         {
1043             // for all panel on the lifting line
1044             forAll(USTAR[i][j], k)
1045             {
1046                 USTAR[i][j][k].x() = 0.5 * (Foam::sqrt(1 + DesiredCT[i])-1) ;
1047             }
1048         }
1049     }
1050 }
```

- With this estimation the β_i is computed in *findTanBetaI()* function. This function computes $\mathbf{V}^*$ first from the current inflow at the current control point. Where $V_t + \omega r$ is already included in `windVectors[i][j][k].y()`.

$$V_a^* = V_a + u_a^* \tag{2.2}$$

$$V_t^* = V_t + \omega r + u_t^* \tag{2.3}$$

β_i follows then at the control points:

$$\beta_1 = arctan\left(\frac{V_a + u_a^*}{V_t + \omega r + u_t^*}\right) \tag{2.4}$$

Since the inflow angle is not the same on the vertices, β_i is interpolated towards the vertex points along the lifting line by the adopted interpolation function *interpolate*. This however

gave very strange results at the blade root and the tip. Hence for the time being, the values for β_i at the outermost vertices are set manually to the β_i values of the last next control point.

Source Code: openPropLiftingLine.C

```
1066 //- Compute tan betaI
1067 void openPropLiftingLine::findTanBetaI()
1068 {
1069     forAll(TANBIc, i)
1070     {
1071         forAll(TANBIc[i], j)
1072         {
1073             forAll(TANBIc[i][j], k)
1074             {
1075                 VSTAR[i][j][k].x() = windVectors[i][j][k].x() + USTAR[i][j][k].x
                        ();
1076                 VSTAR[i][j][k].y() = windVectors[i][j][k].y() + USTAR[i][j][k].y
                        ();
1077
1078                 TANBIc[i][j][k]    = VSTAR[i][j][k].x() / VSTAR[i][j][k].y() ;
1079             }
1080             int iter = 0;
1081             forAll(TANBIv[i][j], k)
1082         // Assumption:  The TANBI is the same at the outermost vortex points
1083         //          as on the control points!
1084         // LATER!       Implement an extrapolation from the control points
1085         //          to the vortex points!
1086             {
1087                 DynamicList<scalar> tmp1;
1088                 tmp1.append(bladeRadius[i][j]);
1089                 DynamicList<scalar> tmp2;
1090                 tmp2.append(TANBIc[i][j]);
1091                 TANBIv[i][j][k]    = interpolate(bladeVortexRadius[i][j][k],tmp1,
                        tmp2);
1092                 tmp1.clear();
1093                 tmp2.clear();
1094                 iter++;
1095             }
1096             // Now, for the moment, set the outermost TANBIv to the outermost
                    TANBIc
1097             TANBIv[i][j][0] = TANBIc[i][j][0];
1098             TANBIv[i][j][iter] = TANBIc[i][j][iter-1];
1099         }
1100     }
1101 }
```

- This was the first step during the iteration function. Next we have to calculate the the horseshoe influence within the *horseShoe()* function. In this function we calculate the axial and tangential velocities induced at the control point of panel k in the following source code, by each of the surrounding panels. According to equations 1.5 and 1.6. Thus the corresponding `UHIFc` lists for each propeller a list for each blade, containing a list for each panel over a list of vectors for its neighbours influence.

Source Code: openPropLiftingLine.C

```
1157 //- Compute horseshoe influence
1158 void openPropLiftingLine::horseShoe()
1159 {
1160     forAll(UHIFc, i)
1161     {
1162         forAll(UHIFc[i], j)
1163         {
1164             forAll(UHIFc[i][j], k)
```

```
1165             {
1166                 // for all vortices find induced velocity at RC by a
1167                 // unit vortex shed by RV(m)
1168                 // wrench returns 2*pi*R*u_bar
1169                 forAll(UW[i][j], l)
1170                 {
1171                     scalar normedRc = bladeRadius[i][j][k]/TipRad[i];
1172                     scalar normedRv = bladeVortexRadius[i][j][l]/TipRad[i];
1173
1174                     UW[i][j][l] = wrench(NumBl[i],TANBIv[i][j][l], normedRc,
                            normedRv);
1175
1176                     // LATER! Include hub-image effects!!!
1177                     // LATER! Include swirl cancellation factor
1178                     // UW[i][j][k].y() = UW[i][j][k].y() * SCF
1179                 }
1180
1181                 forAll(UHIFc[i][j][k], m)
1182                 {
1183                     UHIFc[i][j][k][m] = UW[i][j][m+1] - UW[i][j][m];
1184                 }
1185             }
1186         }
1187      }
1188
1189 }
```

- To compute the neighbours influence we call the function *wrench()* which returns a vector of $\overline{\mathbf{u}}$ (`UW`). This function calculates the equations 1.7 to 1.15, depending on the current neighbours radius.

Source Code: openPropLiftingLine.C

```
1191 //- Compute Wrench influence function
1192 vector openPropLiftingLine::wrench(scalar Z, scalar& TANBIv, scalar& Rc, scalar&
         Rv)
1193 {
1194     scalar y  = Rc/(Rv*TANBIv);
1195     scalar y0 = 1/TANBIv;
1196     scalar U  = Foam::pow(((y0 * (Foam::sqrt(1+Foam::sqr(y))-1))*(Foam::exp(Foam
            ::sqrt(1+Foam::sqr(y))-Foam::sqrt(1+ Foam::sqr(y0)))) / (y*(Foam::sqrt(1+
            Foam::sqr(y0))-1))),Z);
1197     vector uw = vector::zero;
1198     if (Rc == Rv)
1199     {
1200         uw = vector::zero;
1201     }
1202     else if (Rc < Rv)
1203     {
1204         scalar F1 =(-1 / (2*Z*y0)) * (Foam::pow(((1+Foam::sqr(y0))/(1+Foam::sqr(y
                ))),0.25)) * ((U/(1-U)) + (1/(24*Z)*((9*Foam::sqr(y0)+2)/(Foam::pow
                ((1+Foam::sqr(y0)),1.5)) + (3*Foam::sqr(y)-2)/(Foam::pow((1+Foam::sqr
                (y)),1.5)))) * (Foam::log(1 + U/(1-U))));
1205         uw.x() = (Z / (2*Rc)) * (y-2*Z*y*y0*F1);
1206         uw.y() = (Foam::sqr(Z)/Rc) * (y0*F1);
1207     }
1208     else
1209     {
1210         scalar F2 = (1 / (2*Z*y0)) * (Foam::pow(((1+Foam::sqr(y0))/(1+Foam::sqr(y
                ))),0.25)) * ((1/(U-1)) - (1/(24*Z)*((9*Foam::sqr(y0)+2)/(Foam::pow
                ((1+Foam::sqr(y0)),1.5)) + (3*Foam::sqr(y)-2)/(Foam::pow((1+Foam::sqr
                (y)),1.5)))) * (Foam::log(1 + 1/(U-1))));
1211         uw.x() = -(Foam::sqr(Z) / (Rc)) * (y*y0*F2);
1212         uw.y() =  (Z / (2*Rc)) * (1+2*Z*y0*F2);
1213     }
```

```
1214     return uw;
1215 }
```

- Finally the induced velocities $\mathbf{u}^*$ are calculate in the *inducedVelocity()* function, by summing the velocities induced by each horseshoe vortex, as given by equations 1.3 and 1.4.

Source Code: openPropLiftingLine.C

```
1102 //- Compute induced velocities
1103 void openPropLiftingLine::inducedVelocity()
1104 {
1105     vector tmp = vector::zero;
1106     forAll(USTAR, i)
1107     {
1108         forAll(USTAR[i], j)
1109         {
1110             forAll(USTAR[i][j], k)
1111             {
1112                 tmp = vector::zero;
1113                 forAll(UHIFc[i][j][k], m)
1114                 {
1115                     // Interpolate given circulation from input station to
1116                     // bladePointRadius
1117                     scalar G = interpolate(bladeRadius[i][j][m], BladeStation[i],
                                 Circulation[i]) / (2 * Foam::constant::mathematical::pi
                                * TipRad[i] * VS[i]);
1118                     tmp = tmp + UHIFc[i][j][k][m] * G ;
1119
1120                 }
1121
1122                 USTAR[i][j][k] =  tmp;
1123
1124             }
1125         }
1126     }
1127 }
```

- With the new induces velocities, the new total inflow velocity $\mathbf{V}^*$ can be updated by the *findTanBetaI* function and the velocity magnitude `VSTARmag` is calculated according to:

$$V^* = \sqrt{(V_a + u_a^*)^2 + (\omega r + V_t + u_t^*)^2} \tag{2.5}$$

in:

Source Code: openPropLiftingLine.C

```
1051 //- Compute the magnitude of the total inflow velocity
1052 void openPropLiftingLine::computeVSTARmag()
1053 {
1054     forAll(VSTARmag, i)
1055     {
1056         forAll(VSTARmag[i], j)
1057         {
1058             forAll(VSTARmag[i][j], k)
1059             {
1060                 VSTARmag[i][j][k] = Foam::sqrt(sqr(windVectors[i][j][k].x()+USTAR
                        [i][j][k].x()) + sqr(windVectors[i][j][k].y() + USTAR[i][j][k
                        ].y())) ;
1061             }
1062         }
1063     }
1064 }
```

- Now we can calculate the forces on the blade based on the given input circulation and the corrected inflow conditions. This is done on the function *computeBladeForce()*. To do this for all propellers, each blade and each control points, first we need to get the input for the given circulation, drag coefficient and the chord length, interpolated on the control point radii.

Source Code: openPropLiftingLine.C

```
757 void openPropLiftingLine::computeBladeForce()
758 {
759     // Proceed propeller by propeller.
760     forAll(windVectors, i)
761     {

...

773         forAll(windVectors[i], j)
774         {
775             scalar bladeThrust = 0.0;
776             // Proceed point by point.
777             forAll(windVectors[i][j], k)
778             {
779                 // Interpolate the local chord.
780                 scalar localChord = interpolate(bladeRadius[i][j][k],
                        BladeStation[m], BladeChord[m]);
781
782         // Interpolate the local Circulation
783         scalar Gamma = interpolate(bladeRadius[i][j][k], BladeStation[m],
                Circulation[m]);
784                 vector bladeUp = vector::zero;
785                 bladeUp.z() = 1;
786         // Interpolate the local given drag coefficient
787         scalar localDrag = interpolate(bladeRadius[i][j][k], BladeStation[m],
                DragCoeff[m]);
```

- Then these values are taken to calculate the lift and drag per density in local blade coordinates, following equations 1.1 and 1.2. The `F` in the following source code is a factor for tip/root loss, adopted from Churchfield. The lift force is perpendicular to the local $\mathbf{e_r}$ and the total inflow velocity $\mathbf{V}^*$, while the drag is aligned with the inflow but in the opposite direction. Both forces are then transformed back to the global coordinate system of the current propeller and summed as the blade force.

```
819         lift[i][j][k] = F * VSTAR[i][j][k] ^( Gamma * bladeUp);
820         drag[i][j][k] = -VSTAR[i][j][k] * (0.5 * F * Foam::sqr(VSTARmag[i][j][k])
                 * localDrag * localChord);
821
822                 // Make the scalar lift and drag quantities vectors in the
                        Cartesian coordinate system.
823                 vector trueLift;
824                 trueLift.x() = lift[i][j][k] & bladeAlignedVectors[i][j][0];
825                 trueLift.y() = lift[i][j][k] & bladeAlignedVectors[i][j][1];
826                 trueLift.z() = lift[i][j][k] & bladeAlignedVectors[i][j][2];
827
828                 vector trueDrag;
829                 trueDrag.x() = drag[i][j][k] & bladeAlignedVectors[i][j][0];
830                 trueDrag.y() = drag[i][j][k] & bladeAlignedVectors[i][j][1];
831                 trueDrag.z() = drag[i][j][k] & bladeAlignedVectors[i][j][2];
832
833                 // Add up lift and drag to get the resultant force/density
                        applied to this blade element.
834                 bladeForce[i][j][k] = -(trueDrag + trueLift);
```

- The body forces are finally computed in Churchfields *computeBodyForce()* function using a Gaussian projection. Where $f_i(r)$ is the force projected as a body force onto volume grid,

$F_i^{LiftingLine}$ is the lifting line point force, r is the distance between grid cell centre and the control point and ϵ a control factor for the projection.

$$f_i(r) = \frac{F_i^{LiftingLine}}{\epsilon^3 \pi^{\frac{3}{2}}} \exp\left[-\left(\frac{r}{\epsilon}\right)^2\right] \tag{2.6}$$

In the function this is done for each cell in the vicinity of the blade, stored in `sphereCells[i][m]`.

Source Code: openPropLiftingLine.C

```
858 void openPropLiftingLine::computeBodyForce()
859 {
...
876                     // For each blade point.
877                     forAll(bladeForce[i][j], k)
878                     {
879                         scalar dis = mag(mesh_.C()[sphereCells[i][m]] -
                                bladePoints[i][j][k]);
880
881                         if (dis <= smearRadius[i])
882                         {
883
884                             bodyForce[sphereCells[i][m]] += bladeForce[i][j][k] *
                                  (Foam::exp(-Foam::sqr(dis/epsilon[i]))/(Foam::
                                 pow(epsilon[i],3)*Foam::pow(Foam::constant::
                                 mathematical::pi,1.5)));
885
886                         }
```

2.2 Updating your solver

We will go step by step through the procedure to create a solver that can use the new novel class. In this case we will modify the openFOAM *pisoFoam* solver. To start with, you should create a `/src` and an `application` directory in your user directory. Copy the shipped directory of the solver to your user directory for applications in the solvers for propulsion.

```
mkdir -p user-2.1.x/src/propellerModels
mkdir -p user-2.1.x/applications/solvers/propulsion/
cp -r
```

To create an object of the class in the solver *pisoFoam* we need to add the following lines to the *createFields.H* to declare the object:

Source Code: createFields.H

```
48     propellerModels::openPropLiftingLine propellers(U);
```

Next, the class header file needs to be included into the solver code:

Source Code: pisoFoamLLprop.C

```
41 #include "openPropLiftingLine.H"
```

And the body forces needs to be added to the solver momentum equation:

Source Code: pisoFoamLLprop.C

```
67         // Pressure-velocity PISO corrector
68         {
69             // Momentum predictor
```

```
70
71              fvVectorMatrix UEqn
72              (
73                  fvm::ddt(U)
74                + fvm::div(phi, U)
75                + turbulence->divDevReff(U)
76                - propellers.force()
77              );
```

To update the propeller model, include the *update()* function at the end of the time loop:

Source Code: pisoFoamLLprop.C

```
138         Info<< "start propeller update" << endl;
139         //Update the propeller array.
140         propellers.update();
141
142         runTime.write();
```

Now you have changed a part of openFOAM and hence you might want to change the name of the solver to be consistent with your own solver. To do so, write the following in your terminal:

```
cd user-2.1.x/applications/solvers/propulsion/
mv pisoFoam pisoFoamLLprop
mv pisoFoamLLprop/pisoFoam.C pisoFoamLLprop/pisoFoamLLprop.C
```

Alternatively, download the *liftingLineTutorial.tar.gz* from the course homepage and extract it into your openFOAM user directory e.g.

```
tar -xvzf liftingLineTutorial2.tar.gz -C ~/OpenFOAM/user-2.1.x
```

For the compilation the file in the *Make* directory of the solver needs to be modified. Change the Make/files to:

Source Code: Make/files

```
1 pisoFoamLLprop.C
2
3 EXE = $(FOAM_USER_APPBIN)/pisoFoamLLprop
```

Change the Make/options to:

Source Code: Make/options

```
1  EXE_INC = \
2      -I$(LIB_SRC)/turbulenceModels/incompressible/turbulenceModel \
3      -I$(LIB_SRC)/transportModels \
4      -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
5      -I$(LIB_SRC)/finiteVolume/lnInclude \
6      -I$(WM_PROJECT_USER_DIR)/src/propellerModels/lnInclude
7
8  EXE_LIBS = \
9      -L$(FOAM_USER_LIBBIN) \
10     -lincompressibleTurbulenceModel \
11     -lincompressibleRASModels \
12     -lincompressibleLESModels \
13     -lincompressibleTransportModels \
14     -lfiniteVolume \
15     -lmeshTools \
16     -luserPropellerModels
```

Before compiling the solver, we need to compile the class as a customer library:

```
cd user-2.1.x/src/propellerModels
wmake libso
```

Afterwards, we can compile the recently changed solver:

```
cd user-2.1.x/applications/solver/propulsion/pisoFoamLLprop
wmake
```

Chapter 3

Test cases

3.1 Open water

As a first basic test case we are going to employ the new class on propeller in open water condition. In this test case we simulate the flow in a very basic box and model the working propeller in the centre. However, propeller geometry is actually not represented. This makes it very easy to mesh. The only grid that we need to create is a simple box, as depicted in figure 3.1. This can easily be done with *blockMesh*. The *baseLocation* of the propeller will be in the centre of this box. The flow is positive in x-direction with a velocity of $1m/s$, which will be our reference ship speed `VS`. The geometric data, like the chord length and the circulation (figure 3.3) are given for the test propeller *4148*. This is a typical test propeller consisting of a basic shape, without skew or rake. The blade layout of this propeller is shown in figure 3.2 and the specific data are provided in the propeller properties folder under the propellers name. The propeller is a three-bladed propeller with a diameter of 1m. Refer to table 2.2 for the detailed propeller propeller properties.

The general arrangement of the propeller is specified in the *propellerArrayProperties* file (see table 2.1 and the provided file). For the current case only one propeller is listed. This propeller has its *baseLocation* at the coordinates $(5.0, 1, 0.5)$ which corresponds to the boss position (compare figure 2.1). The lifting line is discretized into 20 points which are equally distributed along the radius. The rotational speed is 72.02 RPM. The test case was selected like this, to compare the computed results with the output of OpenPROP (see section 3.1.1). In case several propellers are working, one would give the properties for more propellers in this same manner as for *propeller0*. In this test case the k-ϵ turbulence model is employed but any other turbulence model applicable to the *pisoFoam* solver can be used. The boundary conditions for the velocity are: a fixed value at the inlet, zero gradient at the outlet and a fixed value of zero at the walls.

The provided files will simulate the working propeller, starting from time step 0 until 20 seconds. To run the test case, change to the directory, run blockMesh and the new solver:

```
cd user-2.1.x/run/openWater
blockMesh
piaoFoamLLprop
```

Figure 3.4 shows the result of the simulation for 20 seconds. The plane normal to the x-direction is located at the propeller plane. This orthogonal slice is coloured by the body force magnitude, while the longitudinal slice show the velocity component in x-direction. Figure 3.5 provides a closer look at the propeller plane. This figure includes the body forces provided from the new class.

3.1.1 Comparison with OpenPROP results

The results for the induced velocities, the induced pitch angle β_i and the magnitude of the total inflow V^* are compared with the prediction of OpenPROP for the test case in undisturbed inflow. The differences are shown in figure 3.8. This figure plots the particular differences between the two

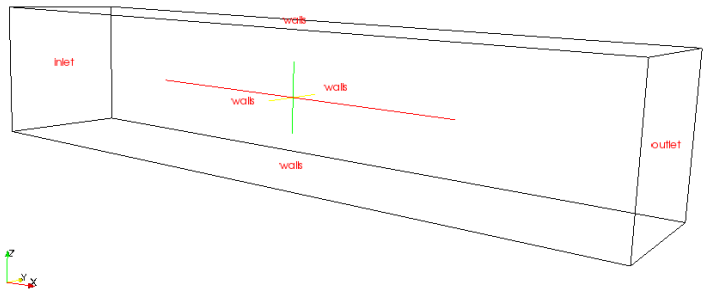

Figure 3.1: Test case 1 set-up

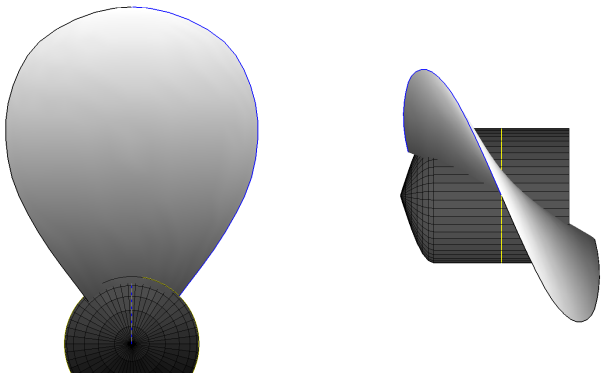

Figure 3.2: Propeller 4118 layout

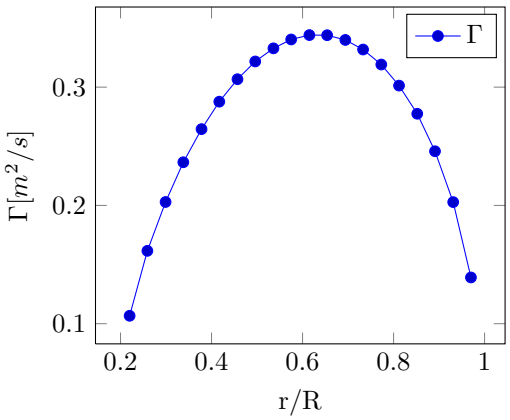

Figure 3.3: Circulation distribution for propeller 4118

codes. One can see that there is in fact a deviation at the root and tip. This might be related to the rather plain interpolation approach. This needs to be revised. However, in general is the result in good agreement, keeping in mind that the effect of the propeller actually effect its inflow due to the introduced body forces.

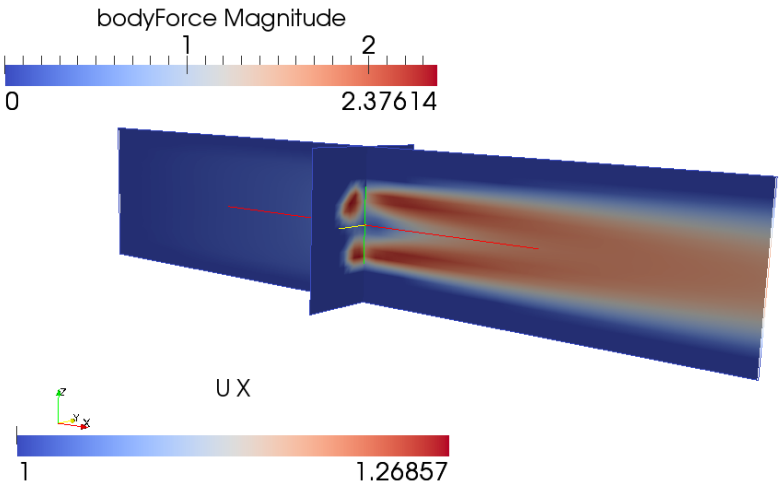

Figure 3.4: Propeller working in the test section

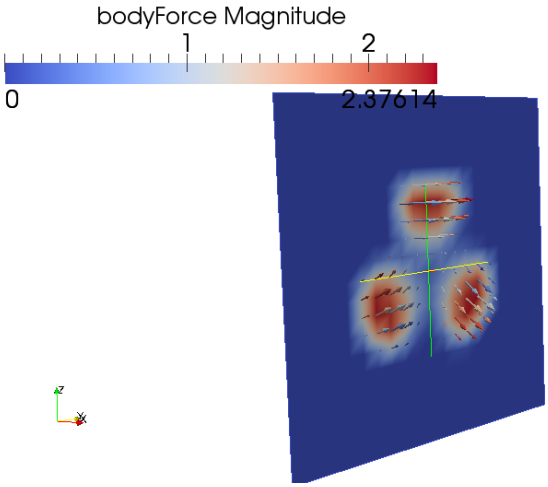

Figure 3.5: Propeller plane with body force vectors

3.2 Artificial wake

In the second test case is the propeller working behind a box. This models a reduced inflow speed to the propeller at the uppermost blade position. This is a rather plain approach and intents only to show the capability of the lifting line class to account for varying inflow conditions. The box is basically taken from the previous example but modified at the inlet to create a slipstream zone. A contour plot of the test section with the developed flow field shown in figure 3.6. The propeller is again placed in the centre of the box and the boundary condition and initial set up is identical to the previous test case. Before the propeller can be activated, the flow needs to develop. This can be done easily with the general *pisoFoam* solver. In figure 3.6 on the left site we can see the velocity distribution in the domain, solved after 10 seconds simulation time. When the flow is developed the new *pisoFoamLLprop* solver is used the include the propeller. There is no need to change any input or to remesh the domain. Only the start time needs to be changed to the last time step of the simulation without the propeller. The result after 15 seconds simulation time can be seen at figure 3.7. The body forces are shown as arrows, the colour scale on the orthogonal slices in shows the velocity component in x-direction. From this figure one can clearly see the reduced acceleration at the upright position as an effect from the blocked inflow.

The provided files contain already the solution for the simulation of 10 seconds without an active propeller. To run 5 further seconds with the active propeller we start from that solution. Since the grid is considerably finer than in the previous test case, it is essential to run parallel. The provided files include the dictionaries for the mesh generation and the decomposition for 4 processors. To run the case, change to the directory, run blockMesh, decompose the domain, run the new solver in parallel and reconstruct the domain after the last time step:

```
cd user-2.1.x/run/Wake
blockMesh
decomposePar
mpirun -np 4 piaoFoamLLprop -parallel
reconstructPar
```

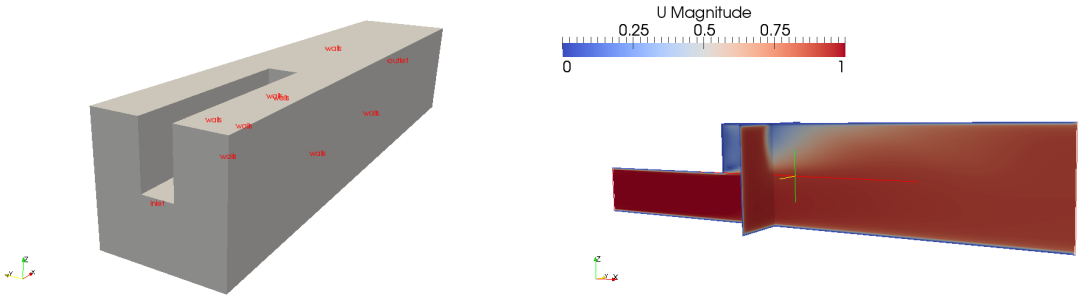

Figure 3.6: Test case 2 propeller in artificial wake. Left: Test section. Right: Developed flow without the propeller

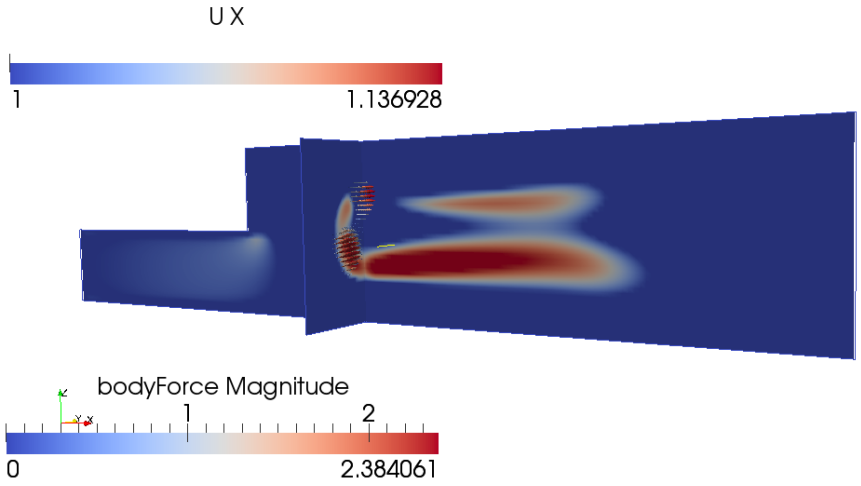

Figure 3.7: Propeller working in artificial wake

3.3 Visualization

The figures in sections 3.1 and 3.2 are created using *ParaView*. For the visualization, open the ParaView with the computed case and select the first time step. In *Object Inspector* click *Apply* to load the case. Once it is loaded at another time step than 0, one can toggle *bodyForce* as a volume field in the *Object Inspector* and use them to colour the a surface or to create vectors. The visualization of the body forces as vectors is accomplished by applying *Glymph* on a slice in the propeller plane.

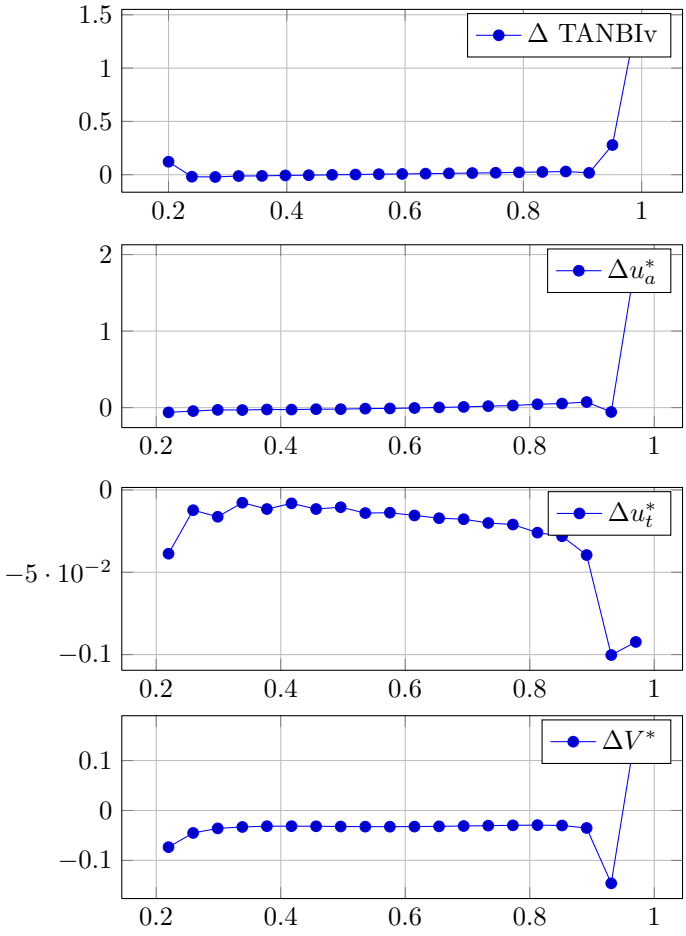

Figure 3.8: Comparison of the results in openFOAM and OpenPROP

Bibliography

[1] Carlton, J. S., 2007, *Marine Propellers and Propulsion (Second Edition)*, Oxford, Butterworth-Heinemann: 164

[2] Chruchfield, M., J., 2011, `http://www.personal.psu.edu/dab143/OFW6/Training/churchfield_slides.pdf`.

[3] Goldstein, S., 1929, *On the Vortex Theory of Screw Propellers*, Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, 123(792): 440-465.

[4] Lerbs, H. W., 1952, *Moderately loaded propellers with a finite number of blades and an arbitrary distribution of circulation*, Trans. SNAME 60.

[5] Epps, B., 2010, OpenPROP *v2.4 Theory Document*

[6] Prandtl, L., 1918, *Tragflügeltheorie I. Mitteilung. Nachrichten von der Königlichen Gesellschaft der Wissenschaft zu Göttingen*, Berlin, Weidmannsche Buchhandlung: 451-477.

[7] Wrench, J. W., 1957, *The calculation of propeller induction factors*, Technical Report 1116, David Taylor Model Basin.