University of Exeter coursework header sheet

ECMM102 Group Project (Meng) (A, TRM1+2 2015/6)

027493

1019571

620006622

Coursework: I2

Submission Deadline: Thu 28th Apr 2016 12:00

Personal tutor: Dr Prakash Kripakaran

Marker name: G Tabor

Word count: 15247

By submitting coursework you declare that you understand and consent to the University policies regarding plagiarism and mitigation (these can be seen online at www.exeter.ac.uk/plagiarism, and www.exeter.ac.uk/mitigation respectively), and that you have read your school's rules for submission of written coursework, for example rules on maximum and minimum number of words. Indicative/first marks are provisional only.

I2 Report

The Development of Free-spinning Wind Turbine Representations for Computational Fluid Dynamics

Benjamin Ian Johnson

2015

4th year MEng Group Project

I certify that all material in this thesis that is not my own work has been identified and that no material has been included for which a degree has previously been conferred on me.

Signed.....  .....

College of Engineering, Mathematics, and Physical Sciences
University of Exeter

I2 Report

ECMM102

Title:

The Development of Free-spinning Wind Turbine Representations for Computational Fluid Dynamics

Word count: 15247

Number of pages: 40

Date of submission: 22/04/2016

Student Name: Benjamin Ian Johnson

Programme: MEng Mechanical Engineering

Student number: 620006622

Candidate number: 027493

Supervisor: Prof. Gavin Tabor

Abstract

Current wind turbine performance predictions are usually based on wind speed data averaged over a period of days or months, however the wind turbine will respond to wind fluctuations in a time-scale of seconds to hours depending on the moment of inertia. There will therefore be some discrepancies between the performance predicted using averaged and fluctuating wind speeds. Existing modelling practices use a prescribed rotational velocity and therefore the model does not respond to the wind fluctuations, consequently this project aims to develop free-spinning modelling techniques capable of responding to the wind conditions. This led to the development of a free-spinning single reference frame (SRF) solver and a free-spinning actuator line model (ALM), both of which were implemented in OpenFOAM. A flexible actuator package capable of having new models and functionality easily written in was created to include the free-spinning ALM, actuator disk models (ADMs) and control systems. The package allows simulations to have any number of turbines and any combination of models and control systems allowing full farm simulations to be set-up without difficulty. The free-spinning SRF was tested on a basic test case and a wind turbine representative case. It performed well on the basic test case and demonstrated the ability to model complex interactions. However on the wind turbine case it was found to be unstable and computationally expensive and it was concluded that it was unviable for use in wind turbine simulations. Conversely the actuator package was found to be very efficient and demonstrated the ability to model free-spinning behaviour without high computational expensive. This package will therefore serve as a platform for future investigations into actuator modelling and the discrepancies between the performance predicted by prescribed and free-spinning turbine models.

Keywords: CFD, Free-spinning, SRF, Actuator Modelling, OpenFOAM, Polymorphism

Table of contents

1. Introduction and background

Computational fluid dynamics (CFD) is frequently used in the design and performance prediction of wind farms; however there are currently large discrepancies between the predicted and realised outputs. This has resulted in a large industry-led investigation into the sources of discrepancy in order to better predict the power output. This project forms part of an undergraduate group project - run in parallel with the industry-led investigation – aiming to evaluate the effects of some causes of discrepancies on the predicted output of wind farms as well as developing some methods and techniques which can later be used to establish the effects of other sources of discrepancies. This project comes under the development section of the group project.

Current estimates of wind farm performance rely on average wind speeds over a time period of days or months, despite wind fluctuations on a time period of seconds to hours. The turbine will respond to wind fluctuations, the exact time-scale of which is dependent on the moment of inertia of the wind turbine. After a substantial literature review, no existing CFD solvers were found which implemented free-spinning modelling techniques, only prescribed velocity solvers exist. Therefore this project aims to develop such techniques in order that a later project may evaluate the discrepancies that the wind fluctuations cause. The two main methods for representing a wind turbine in CFD are moving geometry methods and actuator models. Moving geometry methods have a higher accuracy whereas actuator models have greater flexibility and are cheaper, therefore free-spinning versions of both will be developed.

The literature review also revealed that no actuator package had been developed for OpenFOAM which would enable the free-spinning technique to be implemented without extensive rewriting of the code. Therefore a flexible OpenFOAM actuator package will also be developed to serve as a platform capable of performing all of the necessary simulations that may be required of actuators. Alongside this, the code will need to be tested for functionality.

Section 2 is a review of existing literature relevant to the project. Section 3 covers the theory behind the moving geometry methods and actuator models as well as introducing the cases that will be run to test the code functionality. Section 4 describes the code of the free-spinning Single Reference Frame (SRF) solver and the actuator package as well as giving the results of the test cases. Section 5 is a discussion of the project with recommendations for future work and conclusions drawn from the project. Section 6 covers project management

and sustainability considerations. Section 7 explains the contribution to the overall group project.

2. Literature review

2.1. Moving Geometry

Li et al [1] present a high fidelity approach to wind turbine aero-elastic simulations with explicit representation of the atmospheric wind turbulence. This couples the CFD code to a multi-body dynamics (MBD) code to evaluate the motion responses of the turbine structure to the aerodynamic loading, however the turbine has a prescribed angular velocity. Li et al [2] use detached eddy simulation (DES) and Reynolds averaged Navier-Stokes (RANS) approaches to turbulence modelling to assess their accuracy in wind turbine simulations. Little difference is found in the averaged forces and moments however DES is shown to produce significantly better transient results. Li et al dynamically change the angle of attack and compare to experimental data. Their methodology shows promise for active turbine control simulations and fluid-structure interaction (FSI). However all the simulations are run with a prescribed velocity.

Bazilevs et al [3] present methods of the construction of analysis-suitable wind turbine geometries and a validation of a numerical method is performed on a prescribed velocity SRF rotating domain case. Tran & Kim [4] perform a multiple reference frame (MRF) simulation of a wind turbine, in order to find the impact of the pitching motion of platforms on the wind turbine simulation, for floating off-shore wind turbines (FOWT). Both the pitching motion and rotation are prescribed. In the paper by Sezer-Uzol & Long [5] the case set-up and results of an LES simulation of a wind turbine using a sliding mesh moving geometry method are given. These are validated against experimental results and give good agreement, suggesting that their methodology is accurate. Sanderse et al [6] present a literature review of existing research in the modelling of wind turbines. This is primarily concerned with different actuator models and wake modelling using RANS and large eddy simulation (LES). This paper would therefore be useful to establish the best practices for modelling and which actuator models would be best to implement.

Analysis of the literature of moving geometry CFD found that the majority of research has been to perform modifications to numerics, add in additional calculations or motion or in validating these methods; a small portion of the investigated sources have been reviewed

here. Despite a large research base in FSI of moving geometry simulations, none have been found that extend this to free-spinning aspects. The high computational expense of free-spinning simulations arise due to the long simulated time required. Therefore, this could be the reason as to why little research has been done into free-spinning moving geometry simulations.

2.2. Actuator Modelling

Sørensen and Shen [7] propose an actuator line model (ALM) which uses the blade element method (BEM) coupled to a Gaussian smoothing operation. This model has been widely used and is now the primary ALM used in wind turbine modelling. It has been implemented in the NREL actuator code [8][9] as well as in a spectral-element code [10] and has been well validated [8][10][11][12]. Mikkelson [11] analyses the ALM by Sørensen and Shen [7] in a number of test cases and for coned and yawed turbines. Although the work by Mikkelson will not feed directly into this project, the validation of the models shows that they are accurate and therefore worth implementing.

Martinez Tossas & Leonardi [8] present the NREL actuator code for OpenFOAM. This employs Sørensen and Shen's ALM in a package including an actuator disk model (ADM) which also uses BEM and a Gaussian smoothing function, equivalent to that used by Sørensen and Shen [7] in their ALM. The ADM is commonly referred to as the ADM-R as it includes rotational effects in the wake. Martinez Tossas and Leonardi also provide validation for their code as well as recommended running conditions such as time-steps and cell sizes. The ADM-R is shown to be faster than the ALM and has good agreement for long distance wake effects but fails to capture the tip and root vortices that the ALM can. The NREL code also has control systems allowing the blade and coning angles to be changed. However, the NREL code remains inflexible; with no ability to add in extra models or to modify models to incorporate extra functionality, such as to make them free-spinning. Thus meaning that the NREL code cannot be used as a base for the free-spinning implementation and there is a need for a flexible actuator package. Churchfield [9] offers further information on the NREL code showing that the actuator package is part of a larger package which includes atmospheric boundary layer (ABL) solvers. Nodeland [12] offers further validation of the NREL actuator code against two sets of wind tunnel data and one set of data from a wind farm. This offers further evidence of the accuracy of the NREL code.

Peet et al [10] present an implementation of the Sørensen-Shen ALM in a spectral

element code with focus on the parallelisation aspects; they also offer a validation of their code. The code used was the Nek5000 which has been shown to be very efficient when highly parallelised and is available open source. However as the ALM implementation by Peet et al is not available it would be necessary to implement the ALM and then to implement the free-spinning aspects. The expertise of those involved in this project is in finite volume code and, in particular, OpenFOAM. Therefore Peet et al offer a suggestion for future projects which could focus on an implementation of the work done by this project in a spectral-element code.

In the paper by Svenning [13] an implementation of the Goldstein-Optimum ADM in OpenFOAM is described and validated; the code implementing the ADM is provided with the paper. As part of this literature review, Svenning's code was updated to work with OpenFOAM 2.4 and tested. However it was found to be inefficient and inspection of the code suggested that, in order to implement the volume force, the code was looping through every cell in the mesh to determine the cells in the disk. The volume force was also being recalculated at each iteration of the SIMPLE algorithm, although it did not change between each iteration. Furthermore the code only allows for a single turbine, this was therefore modified by the group to allow for multiple turbines, although the method for calculating each actuator remained the same. This meant that the simulation had inefficiencies of the order NxIxT where N is the number of cells in the mesh, I is the number of iterations of the SIMPLE algorithm and T is the number of turbines. For a cheap simulation of a wind farm this would result in inefficiencies in the order of 10^{11} extra evaluations of functions, increasing to 10^{14} for a detailed case. Svenning's paper has been frequently cited as it is one of the few implementations of actuators in OpenFOAM - thus emphasising the need for a more efficient and more flexible package. In the paper by Jeromin et al [14], Svenning's implementation of an ADM in OpenFOAM is modified to use the blade element method (BEM) in order to model the MEXICO rotor and compare it to an implementation in Fluent, using user defined functions (UDFs), and experimental data.

Research into actuator modelling showed that a large amount of the existing research is in implementing the actuator models into different CFD code. This demonstrates that there is a need for a flexible actuator implementation in which the models can simply be added in rather than requiring large amounts of coding. Other existing research has been in validating the models or the code in which they have been implemented. This suggests that this project does not need to extend to validating the models as well as implementing them. Instead

testing should occur to demonstrate the functionality of the code, but validation of the models should be performed by a later project to accompany the release of the code to the community. No research has been found which extends the actuator models to be free-spinning. The reason behind this is unclear; the computational expense of actuator models is low so they are ideal for free-spinning simulations which require a long simulated time.

3. Theoretical background and testing methodology

3.1. Moving Geometry

There are three key methods for simulating moving geometry in CFD; Multiple Reference Frame (MRF), Single Reference Frame (SRF) and Sliding Meshes. All of these methods are able to be used in OpenFOAM, however it is believed that none have ever been modified to be free-spinning. The various solvers in OpenFOAM all use the angular velocity and solve for position or add the effects of the angular velocity to the fluid in the reference frame. Therefore all of the solvers would take a similar level of modification to add in the effects of the free-spinning geometry.

The SRF solver rotates the reference frame of the simulation and adds in the necessary forces to create the effect of rotating geometry. With the SRF input it is possible to introduce swirl so that the fluid is stationary in the absolute reference frame but relative to the geometry it is swirling. The MRF solver uses the principle of the rotating reference frame but does so in a specific region of the mesh thus allowing a second stationary reference frame. This effect can be used to create multiple spinning regions, whereas an SRF can only model one turbine. However since the reference frame is moving relative to the stationary plane but the geometry is not, an MRF only represents a realistic case when run in steady-state. Otherwise the transient effects of the fluid are being examined in a simulation that is inherently not transient. Therefore the MRF will not be suitable for examining the effects of free-spinning geometry. An alternative to the MRF and SRF is the sliding mesh technique. This physically rotates the mesh about the axis. Interpolation is then used across the sliding interface to account for the non-conformal mesh. Although very accurate, the sliding mesh requires around 10 steps per cell at the interface, causing a large restriction on the time-step. This makes sliding mesh simulations very computationally expensive as it can require thousands of time-steps per rotation. Naturally this makes them unsuitable to be used for free-spinning investigations, as they would require many rotations in order to spin up to speed and would

therefore be too computationally expensive to be practical. Therefore the SRF solver will form the basis of the moving geometry approach to the free-spinning simulations.

For a free-spinning calculation an integral of the pressure at each point over the surface multiplied by the radius of the point from the axis of rotation is used to find the torque. For this project viscous forces will not be considered as they are negligible for the majority of flow regimes. The torque is therefore given by equation 1. Discretising equation 1 allows it to be solved computationally over the mesh patch which is spinning about the axis. It is also possible to convert it to vector form so that it can be solved for any orientation and axis. Thus equation 2 is solved by the CFD code in order to find the torque.

$$T = \int \left[\iint P \, dA \right] \cdot r \, dr \tag{1}$$

$$\vec{T} = \sum_{i=0}^{N} \left[\left(p_i \vec{A}_i \cdot \left(\hat{\boldsymbol{c}} \times \frac{\boldsymbol{r}_i}{|\boldsymbol{r}_i|} \right) \right) \left(\hat{\boldsymbol{c}} \times \frac{\boldsymbol{r}_i}{|\boldsymbol{r}_i|} \right) \times \boldsymbol{r}_i \right] \quad ; \quad \boldsymbol{r}_i = \boldsymbol{m}_i - \left(\boldsymbol{c}_s + \left(\left(\boldsymbol{m}_i - \boldsymbol{c}_s \right) \cdot \hat{\boldsymbol{c}} \right) \hat{\boldsymbol{c}} \right) \tag{2}$$

Where: N is the number of cells in the mesh patch, p_i is the pressure, A_i is the area vector, $\hat{\mathbf{c}}$ is the axis unit vector, $\mathbf{r}$ is the radial vector from the axis to the mesh point, $\mathbf{m_i}$ is the vector of the mesh point coordinates and $\mathbf{c_s}$ is the vector of the axis start point coordinates. The torque is related to the angular acceleration, α, by the moment of inertia, I, as follows in equation 3. The angular velocity is then given by the integral of the angular acceleration with respect to time. For this case a numerical integration scheme must be used, therefore a forward Euler (also referred to as upwind differencing) scheme will be used as this is both fast and stable. Thus the angular velocity at the next time-step, ω_{n+1}, is given by equation 4, where Δt is the time-step.

$$T = I \, \alpha \quad \Rightarrow \quad \alpha = \frac{T}{I} \tag{3}$$

$$\omega_{n+1} = \omega_n + \alpha \, \Delta t \tag{4}$$

One possible modification to this would be to use an Adams-Bashforth and Adams-Moulton predictor-corrector method for time-stepping the angular velocity. This would add extra accuracy and stability without the requirement for intermediate evaluations, such as with the Runge-Kutta schemes. This is a possible addition to the code, although the effect on the duration of the runs and the solution should be evaluated fully. The SRF solver uses an angular velocity to modify the reference frame to account for the rotation. Therefore once the angular velocity has been calculated, the modifications to the SRF will only need to add in the ability to set the angular velocity and the solver will then be free-spinning.

3.2. Actuator Modelling

3.2.1. Actuator Disk Models

An ADM applies a force on the fluid in a disc region. The models can add in both an axial and a tangential force to replicate the velocity deficit and the swirl of a wind turbine wake. The ADM does not model the effects of the individual blades but instead models the whole swept area, therefore the wake near to the disk will not be accurate compared to a real turbine but the far-wake effects can be. It is for this reason that they are commonly used in farm simulations. Clearly because an ADM does not model the blades it cannot be used for free-spinning calculations investigating the near wake effects. However some models apply axial and tangential forces based on the velocity relative to the blades, therefore these models can have a free-spinning aspect to them. Since part of this project is to produce an actuator package which will incorporate all the necessary elements to investigate free-spinning simulations, this package should include ADMs so that they can be used to produce the effects of upwind turbines. The two ADMs that will be used are the Goldstein-Optimum Model and the ADM-R model as these have both been previously used by Svenning [13] and the NREL [8] respectively and are well validated. The Goldstein-Optimum Model applies forces that vary radially with the form [13]:

$$f_x = A_x r^* \sqrt{1 - r^*} \tag{5}$$

$$f_\theta = A_\theta \frac{r^* \sqrt{1 - r^*}}{r^* \left(1 - r'_h\right) + r'_h} \tag{6}$$

$$r^* = \frac{r' - r'_h}{1 - r'_h} \quad ; \quad r' = \frac{r}{R_t} \quad ; \quad r'_h = \frac{R_h}{R_t} \tag{7}$$

Where f_x is the axial force, f_θ is the tangential force, R_h is the hub radius and R_t is the turbine radius. The constants A_x and A_θ are calculated so that the force over the whole volume of the actuator disk adds up to the total prescribed axial force, F_x, and torque, F_θ, as in equations 8 and 9 [13]. Where b is the disk thickness.

$$A_x = \frac{105}{8} \frac{F_x}{\pi b \left(3R_h + 4R_t\right)\left(R_t - R_h\right)} \tag{8}$$

$$A_\theta = \frac{105}{8} \frac{F_\theta}{\pi b R_t \left(3R_t + 4R_h\right)\left(R_t - R_h\right)} \tag{9}$$

To calculate the prescribed axial force and torque it is possible to use the thrust and power coefficients, which are usually known for wind turbines. The torque can be found since power, W, is equal to the torque multiplied by the angular velocity, ω.

$$F_x = \frac{1}{2} C_F \rho \pi R_t^2 V^2 \tag{10}$$

$$W = \frac{1}{2} C_w \rho \pi R_t^2 V^3 \quad \Rightarrow \quad F_\theta = \frac{1}{2\omega} C_w \rho \pi R_t^2 V^3 \tag{11}$$

The ADM-R model that is implemented by the NREL uses an evaluation of the lift and drag coefficients to get the force on the fluid of a 2D blade element. This is then smoothed out across the fluid by using a Gaussian smoothing function, as proposed for an ALM by Sørensen and Shen [7]. The lift and drag are both evaluated for each blade element based on the local fluid angle and relative velocity as shown in Figure 1. The local fluid angle is given by $\alpha - \theta$, whereby θ is the local angle of attack of the blade section and α is the relative fluid angle. V_a is the axial velocity.

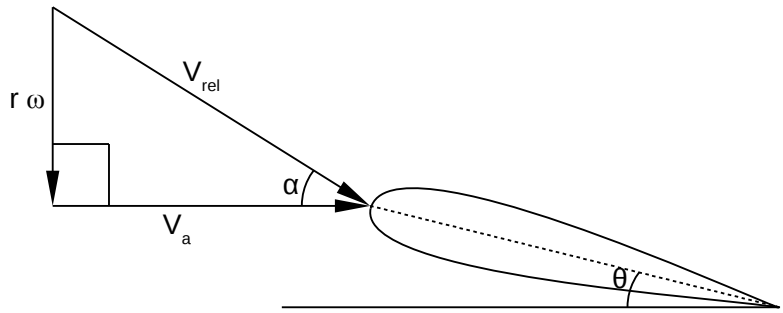

Figure 1: Diagram of the velocity relative to the blade section.

The 2D blade force is then given by equation 12, where c is the chord length and $\hat{\boldsymbol{e}}_L$ and $\hat{\boldsymbol{e}}_D$ are unit vectors in the directions of lift and drag. The direction of the lift force on the fluid is opposite to the lift on the blade and normal to the relative velocity. Whereas the direction of the drag force on the fluid is opposite and parallel to the relative velocity.

$$\vec{f}_{2D} = \frac{1}{2} \rho c V_{rel}^2 \left(C_L \hat{\boldsymbol{e}}_L + C_D \hat{\boldsymbol{e}}_D \right) \tag{12}$$

The Gaussian smoothing function is then applied along the centreline. This acts to remove singularities and create a more realistic effect on the fluid. The Gaussian smoothing function is shown in equation 13, where ε establishes the width of the smoothing function and the maximum value and d_b is the distance in the centreline direction of the point from the mid-plane of the disc.

$$\eta_\varepsilon = \frac{1}{\varepsilon^3 \pi^{3/2}} \exp\left[-\left(\frac{d_b}{\varepsilon} \right)^2 \right] \tag{13}$$

The Gaussian smoothing function and 2D blade forces are calculated for each point and multiplied together to give the volume force. The NREL code also includes a solidity factor which which is equal to the number of blades multiplied by the area of each blade divided by the swept area of the blades. This is used to account for differences in blade numbers and sizes. It should be clear that because the volume force of the ADM-R is based on the relative velocity it is possible to incorporate a free-spinning calculation into the model in order to replicate the far-wake of a free-spinning turbine. A similar approach could be made with the Goldstein-Optimum model but would require lookup tables of the coefficients of thrust and power for each angular velocity.

3.2.2. Actuator Line Models

The most commonly used ALM is the Sørensen-Shen model [7] this uses an evaluation of the 2D blade force as in equation 12 in section 3.2.1, which has the same calculation of the relative fluid velocity and local fluid angle as in Figure 1. This 2D blade element force is then smoothed radially outwards from each blade using a Gaussian smoothing function as in equation 14, where r is the radius from the blade section [7][8].

$$\eta_\varepsilon = \frac{1}{\varepsilon^3 \pi^{3/2}} \exp\left[-\left(\frac{r}{\varepsilon}\right)^2\right] \quad \textbf{(14)}$$

The value of ε can be tuned to match experimental data, however a value of ε less than twice the grid-scale can introduce instability [8]. The Gaussian smoothing function is applied multiplicatively, therefore the body force of the blade on the fluid is given by equation 15 [8]. It should be clear that the Sørensen-Shen ALM is very similar in construction to the ADM-R.

$$\vec{f}_\varepsilon = \frac{\eta_\varepsilon}{2} \rho c V_{rel}^2 \left(C_L \hat{\boldsymbol{e}}_{\boldsymbol{L}} + C_D \hat{\boldsymbol{e}}_{\boldsymbol{D}}\right) \quad \textbf{(15)}$$

3.2.3. Free-spinning Actuator

For the actuator models that use blade elements with different properties at each section, a lookup table is used to define the properties along the blade. These include the coefficients of lift and drag, the local angle of attack of the blade and the chord length, among other properties. A linear interpolation is used to find the intermediate values between the intervals of the specified blade properties. The forces are evaluated for the mid-point of each interval and used to find the torque of that section. If local velocity sampling is used this will give a series of points along the blade at which the velocity is known. Therefore in order to get an accurate approximation of the torque on the turbine, the smallest interval must be used.

The torque on the turbine is given by equation 16, where N is the number of blade sections, N_b, or the number of velocity sampling intervals along the blade, N_v, whichever is greatest. Equation 17 shows the evaluation of the local 2D force on the section, the chord length and the coefficients of lift and drag are interpolated from the blade properties. The relative velocity is interpolated from the velocity sampling regions. If lookup tables are used to find find the coefficients of lift and drag based on the local fluid angle, then this is calculated from the interpolated velocity.

$$T_{blade} = \sum_{i}^{N} \vec{f}_{2Di} \cdot r_i \quad ; \quad N = max(N_b, N_v) \tag{16}$$

$$\vec{f}_{2Di} = \frac{1}{2} \rho c_i^I (r_2 - r_1) V_{rel,i}^{I\ 2} \left(C_{L,i}^I \hat{\boldsymbol{e}}_{\boldsymbol{L,i}} + C_{D,i}^I \hat{\boldsymbol{e}}_{\boldsymbol{D,i}} \right) \tag{17}$$

$$x_i^I = \frac{x_2 - x_1}{r_2 - r_1} \cdot r_i + x_1 \quad ; \quad r_i \in [r_1, r_2] \tag{18}$$

This procedure is repeated for each blade to find the total torque on the turbine. The re-evaluation of the torque for each blade, rather than multiplying by the number of blades, is due to the local velocity sampling. Once the torque on the turbine is found, the angular acceleration can be calculated from $T = I\alpha$, then a forward Euler numerical integration is used to find the angular velocity, as with the SRF model in section 3.1. Rotation of the blades is then achieved via multiplying the angular velocity by the time-step to get the angle. Rodrigues Formula [15] is then used to rotate the vector defining the blade. This formula is shown in equation 19, where $\hat{\boldsymbol{a}}$ is the unit vector about which the rotation occurs and θ is the angle of the rotation.

$$\boldsymbol{v}' = \boldsymbol{v}\cos(\theta) + \hat{\boldsymbol{a}}(\hat{\boldsymbol{a}} \cdot \boldsymbol{v})(1 - \cos(\theta)) + (\hat{\boldsymbol{a}} \times \boldsymbol{v})\sin(\theta) \tag{19}$$

3.3. Code Theory

3.3.1. Object Orientation

There are two main types of coding language, procedural code and object orientated code. Procedural code consists of a series of functions and commands whereas object orientated code allows the creation of objects belonging to classes, where the object has functions belonging to it. OpenFOAM is written in C++ which is object orientated. The main benefits of object orientation are in code organisation and flexibility. An object orientated code consists of the application, which is the main code that is run, and the object classes, which define the attributes and functions that the objects own. In C++ a class consists of a

header file, which declares the class and all the attributes and functions within it, and a code file, which defines the functions.

Classes enable a hierarchical structure whereby a base class can be created which child classes inherit from, thus gaining access to the functions and attributes. In C++ this inheritance is controlled in the class definition in the code header file, where the tags of public, protected and private are used to designate what can access the attributes or the functions. Anything with access to the object belonging to the class can also access the public elements, however only children of the class can access the protected elements and only the object itself has access to the private elements. Careful management of the inheritance of attributes and functions ensures that classes do not overwrite the wrong variables and function definitions do not replace others. When a class is inheriting from another, the inheritance is designated public, protected or private to define what has access to the inherited elements.

3.3.2. Pointers

Pointers are a key feature of C++ and are critical for memory efficiency and flexibility, including polymorphism (see section 3.3.3). Instead of creating an instance of an object in the current workspace, it is possible to instead create a pointer to one. When called, a pointer returns the memory location of the object; by dereferencing a pointer, the object to which it points is accessed. Therefore a pointer to an object can first be dereferenced and then a member function of the object can be called. Two key benefits of a pointer are that it can be redirected to point at another variable and that it can be used to point to nothing – useful in initialisations or for optional parameters. When passing a variable to a function, a copy of it is created unless the variable is passed by reference or by pointer. Creating a copy is inefficient particularly in CFD as the memory use of some objects can be many gigabytes.

To create a pointer to a new object the keyword “new” is used. This creates an instance of the object on the heap, outside of the current workspace, which can only be accessed through the pointer. This can cause memory leak problems if the object is not deleted correctly. One benefit of pointers is that they offer an elegant way of breaking cyclic relations in code, for example: Within a class, a special pointer called “this” can be used to return a pointer to the object belonging to the class. It is the flexibility of the pointer allowing it to be redirected to target another variable or object that is key to dynamic allocation and full polymorphism.

3.3.3. Polymorphism

The complex hierarchical structures that can be created in C++ enables polymorphism to be used. Take a base class from which a number of sub-classes are derived and each of the sub-classes have the same functions that the base class has, but the functions themselves do different tasks. An object could be created which belongs to any of the sub-classes so that calling a function, that all the sub-classes have, will do a task specific to the sub-class the object belongs to. The code itself could therefore change which class of object it is dealing with in order to achieve a different outcome – this is polymorphism. Polymorphism allows some functions which are common to all the derived classes to belong to the base class, however any function that is redefined in the sub-class must be declared as virtual in the base-class. If a function is virtual it means that the program will take whichever function definition is lowest in the hierarchy. Functions which are declared in the base class but are only defined in the sub-classes must be made into pure virtual functions by replacing their definition with =0. This has the effect of making the base class abstract, whereby an instance of the base class can no longer be created however a pointer to the base class can be.

An abstract base class has the use of allowing full polymorphism. One benefit of this is that containers can be created which hold pointers to the base class; therefore, by inheritance, allowing pointers to the sub-classes as well. Thus the container can have pointers to any combination of the base and sub-classes. Another benefit is in dynamic allocation, whereby the pointer can be reallocated to another object of the same base-class. At compile-time a container of pointers to the base class could be created, however at run-time this would reallocate the pointers to a sub-class depending on user input thereby changing the manner in which the code operates. In CFD this is often used for changing the models used, in this case actuator models. The flexibility that polymorphism offers allows additional classes to be easily written into the code to be used at run-time.

3.4. CFD Test Cases

3.4.1. Free-spinning SRF

The free-spinning SRF will need to be tested initially on a very basic and cheap case in order to establish whether the free-spinning aspects are working as expected. It will then require a further test case which will be more computationally expensive, but more representative of a wind turbine. The more expensive case aims to test the viability of a free-spinning SRF investigation of wind turbines.

The basic test case uses the SRFPimpleFoam tutorial case files from OpenFOAM. This is of a vertical axis paddle wheel in a cross flow of 1 m/s with a pseudo-2D circular domain with a radius of 0.2m. The domain has a structured mesh of 2880 cells, making it very computationally cheap. The k-ε turbulence model was used as this gives reasonable performance for external aerodynamics and the accuracy of the simulation was not critical as only the general behaviour was being examined. The free-spinning aspects of this test case were to start the paddle wheel spinning at 60rpm and to leave it to decelerate in the flow. The paddle wheel is a basic cross shape with four paddles. The general trend of the rotor will be to slow down and settle into an oscillatory motion. This behaviour would only occur if the case is free-spinning.

The complex test case is of the axial flow over a test turbine geometry which consisted of three straight angled blades, as this offered a relatively cheap geometry to mesh. The turbine had a blade length of 7m and was situated in a cylindrical domain with radius 40m and length 192m, the turbine was positioned 44.8m from the inlet. The characteristic length (L) of the turbine was taken to be the diameter of the swept area (15m), therefore giving a clearance to the radius of 2.7L, with 2.98L in front and 9.8L behind the turbine. Therefore the fluid should have settled back to far-field behaviour by the time it crosses the boundaries. The domain would require initialisation before being left to freely spin so as to give a reasonable initial torque. The boundary conditions used were SRFFreeStreamVelocity on the cylinder walls to give a swirling far-field boundary, this was later changed to slip for stability; SRFVelocity at the inlet, in order to add in the swirl; wall conditions on the turbine and a pressure outlet. An adjustable time-step was used to ensure a Courant-Friedrichs-Lewy (CFL) number of 1. The PIMPLE algorithm was set to have 50 correctors loops with residual controls to exit the iterations when the residuals dropped below $1x10^{-5}$. The k-ε turbulence model was used initially but later replaced by laminar flow to reduce cost, as the viability not the accuracy was under scrutiny.

3.4.2. Actuator Package

The actuator package required testing of all aspects of the code, therefore a large number of test cases were required. The aspects to test were the individual models, the free-spinning behaviour, the control system, varying the number of blades, varying the number of turbines and different combinations of turbines, models and rotational speeds. This number of test cases was due to the requirement to ensure that all aspects of the code were functioning

as expected. The free-spinning test cases consisted of spinning up to speed and decreasing in speed as this is the most basic behaviour that is expected and gives the best chance of detecting any issues due to predictable behaviour. The implemented control systems currently are only basic angular velocity controls so these will be tested along with the free-spinning test cases. The other test cases will include running a maximum of two turbines (although there is no limit to the number in the code).

The spinning up to speed cases were performed in a wind-speed of 5 m/s and 20 m/s, two cases were run for each wind-speed, both with and without control systems. This allowed the control systems to be checked to ensure that they operate as required. However, it was also important to allow the turbines to spin up to their maximum speed without the control system as this would demonstrate that the model is being evaluated correctly. As the turbine accelerates the forces rotate about the blade until the lift force is no-longer fully driving the turbine but instead has an axial component; the drag force then balances the circumferential component of lift preventing further acceleration. The slowing down cases were set-up so that the turbine had an initial velocity but for one case was in still air while the other was in a 1m/s wind. This causes the turbine to drop in speed until it approaches the velocity where the drag forces responsible for the deceleration are countered by the torque on the turbine.

The test turbine representation has a radius of 2.5m, as with the SRF case the characteristic length-scale (L) was taken to be the turbine diameter. The test cases were run in a cylindrical domain with a radius of 10m (2L) and a length of 80m (16L) the actuator was situated 20m (4L) from the inlet, leaving 60m (12L) for the wake to dissipate before the outlet. A structured mesh was used and had 735875 cells, this used an O-H grid at the cylinder ends to give smaller high quality cells in the cylinder centre, a grading was used to blend the fine cells into the large far-field cells. In the streamwise direction, the cell size was decreased around the actuator location, to give a better definition of the turbine, and in the wake region, to improve flow predictions. A structured mesh was chosen as this ensures very high quality cells which improves the accuracy and run-time of the simulation; the use of structured meshing is one key benefit of using actuators. The boundary conditions used were a 5m/s inlet velocity, slip conditions on the outer walls and a pressure outlet. Again the k-ε turbulence model was used to give basic turbulence predictions.

4. Description of developed code and test results

4.1. Free-spinning SRF

4.1.1. OpenFOAM SRF Implementation

In OpenFOAM the SRF is implemented as a class called SRFModel within the namespace “SRF” within OpenFOAM's namespace “Foam”. The SRFModel class handles the forces and additional physical effects of the SRF. It is used in the solution algorithm to add in a source term to the relative velocity equation as well as to provide the velocity of the SRF which is added to the relative velocity to get the absolute velocity. At the start of the application createFields.H is called which creates the necessary fields of the variables that are being solved for; it is here that a pointer to the current SRFModel is initialised by the lines:

```
autoPtr<SRF::SRFModel> SRF
(
    SRF::SRFModel::New(Urel)
);
```

The function SRFModel::New looks up the model specified in the SRFProperties dictionary and returns a pointer to the SRFModel. The autoPtr is a templated type which has many of the properties of a standard pointer but has benefits in memory management. The SRFModel class has an attribute for the angular velocity (omega_) which it uses in the calculations of the forces and absolute velocity. Ordinarily this attribute is read in from the properties dictionary at the start of the simulation and remains constant. However, it is possible to modify this by adding in a function to the SRFModel that allows the value of omega_ to be set to a specific value. This was done by adding in the declaration of the function to the header file and the definition to the code file as:

```
//Header File:
void setOmega(dimensionedVector newOmega);

//Code File:
void Foam::SRF::SRFModel::setOmega(dimensionedVector newOmega)
{
  omega_ = newOmega;
}
```

This function is called with a dimensioned vector of the angular velocity, where each component of the vector is the rotation about that direction. This function is called after the PIMPLE iterations and the evaluation of the torque. This ensures that the solution is converged for that time-step before adjusting the rotational speed. The next time-step will then use the new value of the angular velocity.

4.1.2. Torque Calculation

A function called computeTorq was written in the application file to calculate the torque on the mesh patch. Initially the function obtains the patchID tag from the mesh object.

```
label patchID = mesh.boundaryMesh().findPatchID(patchName);
```

The patchID is used to access the values at the cell faces of a boundary field, see below code. If the patch does not exist the patchID is set to -1; an if statement is used to check this and to exit if found true. The torque is then initialised as a dimensioned vector with zero value and the specified mesh patch (torquePatch) is identified. Following this the centreline unit vector (clUnitVector) and a vector field of the radial vector from the centreline to the point are calculated as:

```
const fvPatch& torquePatch = mesh.boundary()[patchID];

vector clUnitVector = (clEndPoint-clStartPoint)/mag(clEndPoint-clStartPoint);

vectorField radialVectorFromCL =
    torquePatch.Cf()
  - (
        clStartPoint
      + (
            (
              (torquePatch.Cf() - clStartPoint)
            & clUnitVector
            )
          * clUnitVector
        )
    );
```

By creating the radialVectorfromCL as a vector field, the field operations of OpenFOAM and C++ can be used for speed and code efficiency. The torquePatch is required to be an instantiation of the the boundary field as the member functions of it need to be accessed, which cannot be done on the return value of a function. For example the following line is not valid: `mesh.boundary()[patchID].Cf()` as the boundary field is returned from the function but does not exist in the current workspace until another variable is set as equal to it. The Cf function returns the cell face centres as a vector field, therefore the equation used to find radialVectorFromCL is equivalent to the value of r in equation 2. This performs the operation on all of the values in the vector field in a single line of code.

OpenFOAM uses dimensions at the field level but not at the access level, therefore when accessing values from the pressure field they return dimensionless. In order for the code to operate it is therefore necessary to create a dimensioned variable (dimConvert) with

unit value to convert the torque equation into the dimensions required by the torque variable. The final line of the computeTorq function is therefore:

```
return torque =
  - dimConvert
  * sum
    (
        (
            p.boundaryField()[patchID]*torquePatch.Sf()
          & (clUnitVector ^ radialVectorFromCL/mag(radialVectorFromCL))
        )
      * (clUnitVector ^ radialVectorFromCL/mag(radialVectorFromCL))
      ^ radialVectorFromCL
    );
```

The Sf function of the mesh object returns a vector field of the area vectors of the faces in the patch. The return function indicates the end of the function at which point it passes the value of torque back to the main workspace. Therefore this code is equivalent to using equation 2 (from section 3.1), where radialVectorfromCL is **r**.

$$\vec{T} = \sum_{i=0}^{N}\left[\left(p_i\vec{A}_i\cdot\left(\hat{c}\times\frac{\boldsymbol{r}_i}{|\boldsymbol{r}_i|}\right)\right)\left(\hat{c}\times\frac{\boldsymbol{r}_i}{|\boldsymbol{r}_i|}\right)\times\boldsymbol{r}_i\right] \quad ; \quad \boldsymbol{r}_i = \boldsymbol{m}_i - \left(\boldsymbol{c}_s + \left(\left(\boldsymbol{m}_i - \boldsymbol{c}_s\right)\cdot\hat{c}\right)\hat{c}\right) \qquad \textbf{(2)}$$

4.1.3. Angular Velocity Calculation

After the PIMPLE loop the application calculates the torque on the patch, which is assigned to torqueAboutCL; after this the angular velocity is calculated. A friction torque can be introduced to act against the angular velocity and is added to the torque by:

```
torqueAboutCL -=
    frictionTorqueMag * unitCentreLine
  * angularVelocity.value()/mag(angularVelocity.value());
```

By multiplying by the angular velocity divided by its magnitude this ensures that the friction torque acts against motion. The angular acceleration is then calculated and, if applicable, limited before the angular velocity is updated (see code below). The angular acceleration limit can be set as zero for a prescribed angular velocity simulation or for initialisation purposes. The old value of the angular velocity is stored to calculate the new angular velocity by a forward Euler numerical integration. The additional function that was added to the SRFModel class is then called to update the angular velocity used by the model.

```
angularAcceleration = torqueAboutCL/momentOfInertia;

if (mag(angularAcceleration) > accelerationLimit)
{
    angularAcceleration =
    angularAcceleration/mag(angularAcceleration) * accelerationLimit;
}
```

```
angularVelocity = oldAngularVelocity + angularAcceleration*timestep;
oldAngularVelocity = angularVelocity;
SRF->setOmega(angularVelocity);
```

4.1.4. Input and Output

The input and output (IO) of the FreeSpinningSRFPimpleFoam application has a few key differences from the standard SRFPimpleFoam application. In particular the inputs are read in through the freeSpinControl file in the constant directory. This location was chosen as the file represents the controls to the free-spinning simulations as well as the properties of them. FreeSpinningSRFPimpleFoam also has different outputs than the standard simulation due to the additional parameters of the torque and angular velocity that are calculated; these are both outputted to csv files named “torque” and “omega” respectively at each time-step. This ensures that the free-spinning data can be analysed without requiring a full writing of the simulation data at each step. This output was achieved through the following lines:

```
fileName omegaFile(runTime.path()/"omega");
fileName torqueFile(runTime.path()/"torque");
...
std::ofstream outputFileTorque(torqueFile.c_str(),ios_base::app);
outputFileTorque << torqueAboutCL.value() << "," << std::endl;
outputFileTorque.close();

std::ofstream outputFileOmega(omegaFile.c_str(), ios_base::app);
outputFileOmega << angularVelocity.value() << "," << std::endl;
outputFileOmega.close();
```

As this style of outputting data is not specific to OpenFOAM the standard C++ library can be used. Indeed, due to OpenFOAM's lack of user documentation it is very difficult to find how this would be done using OpenFOAM's library, whereas the C++ library is well documented from a user standpoint. OpenFOAM is built using C++, so a version of C++ is distributed with it thus using the standard library will not cause any dependency issues.

4.1.5. Further OpenFOAM Modifications

The FreeSpinningSRFPimpleFoam solver was run using the PIMPLE algorithm with a high number of corrector loops (~50) with residual control to exit the corrector loop when converged; the settings recommended by OpenFOAM for large potentially unstable cases. However, when running, it was noticed that there was a bug in the PIMPLE algorithm implemented by OpenFOAM. The residuals would remain very low when starting the next time-step, in comparison, many other CFD software suites have residuals of around 1 when starting the next time-step – as the reference values change to the current values. The effect of the residuals not increasing in this way is that the residuals reach a level whereby they fulfil

the convergence criterion to end the corrector loop before starting the iterations. This causes the simulation to skip over the time-step and move onto the next without iterating, which causes severe convergence difficulties. This could be design intent, however, there are guards in the PIMPLE code to ensure that some iteration occurs on each time-step. These guards fail to work and the iterations continue to be skipped over. The reason this is most certainly a bug, is that the PIMPLE algorithm continues to count the progressing time-steps as corrector loops, when the limit is reached it displays the message saying that the corrector loops failed to converge within the specified maximum number.

In order to rectify this bug, the PIMPLE algorithm was modified to add in the addition of a minimum number of corrector loops. This is specified in the same location as the rest of the PIMPLE controls in the fvSolution file. When the residuals drop below the convergence criterion the additional criterion of the minimum number of corrector loops must still be satisfied before the algorithm will progress. This modification was made in the pimpleControl file, that the pimple algorithm is defined in, in the “loop” function. Here, the if statement that checks whether the solution is completed or converged was modified to:

```
if ((converged_ || criteriaSatisfied()) && corr_ >= nCorrectorsMinPIMPLE)
```

A further issue that was experienced was OpenFOAM's handling of the adjustable time-step. This has no minimum value and so, if the simulation is not converging, the time-step will continue to decrease to a point that is completely unrecoverable, rather than allowing the simulation to fail, which would free up the computer resources and alert the user to the problem. Other CFD software packages use a minimum time-step which allows the simulation to decrease to a point after which there is period in which the simulation may recover, if not, it will fail. This is a more elegant way of handling the problem and can be implemented by adding an input (minDeltaT) to the controlDict file and changing the setDeltaT.H file so that it takes a maximum of the calculated value and the minimum value as:

```
runTime.setDeltaT
    (
        max
        (
            min(deltaTFact*runTime.deltaTValue(),maxDeltaT),
            minDeltaT
        )
    );
```

4.1.6. Free-Spinning SRF Code Discussion

The free-spinning SRF code fitted easily into the already implemented SRF in OpenFOAM with very little modification, thus making it very code-efficient. The code however requires some tidying after better coding practices and methodologies were learnt from the coding of the actuator package. This includes correcting the code to fit with the C++ standard coding format and adjusting some of the code to improve efficiency. Specifically the adjustable time-step modification was made in the application file but could easily be modified to take effect in the original file. The torque calculation function also duplicates a number of variables unnecessarily, so this can be adjusted to improve speed and memory efficiency. Although the code could be written to give the SRF control systems and some of the flexibility that the actuator package has, the SRF was found to be too computationally expensive to be viably used for full investigations and would only be able to be used on a single turbine. Therefore, as the uses of the free-spinning SRF are limited, further work on the SRF is not recommended.

4.2. Actuator Code Package

4.2.1. Code Overview

The actuator package code has been developed with a polymorphic structure to allow different models and control systems to be implemented without the need to rewrite large portions of code. This is done via abstract base classes and pointers so that dynamic allocation of classes can occur at run-time, see sections 3.3.2 and 3.3.3. The code required an unlimited number of turbines able to be modelled at the same time, with no restrictions on the combinations of models. The code should also have both ADMs and ALMs already implemented as this would cover the two main actuator modelling groups. Control systems were also required to be implemented as this is another key feature of turbine modelling. The structure of the code is as shown in Figure 2, the solid lines represent inheritance whereas the dotted lines represent dependence. The application actuatorPimpleFoam is at the top of the structure and only dependent on the actuator classes. The actuator classes consist of an abstract actuator base class which has two child classes actuatorLine and actuatorDisk. The actuatorLine class is dependent on the bladeModel classes and the turbineControl classes, whereas the actuatorDisk class is dependent on the turbine control classes and the diskModel classes. At run-time a pointer to the actuator class is allocated to either an actuatorLine or an actuatorDisk. If an actuatorLine is selected a member pointer to the bladeModel base class is reallocated to one of the child classes, in this case the only child is the sorensenShen class. If

an actuatorDisk is selected, then the member pointer to the diskModel base class is reallocated to a child of the diskModel, in this case either the goldsteinOptimum or admR classes. The same process occurs for both actuatorLine and actuatorDisk with the turbineControl class.

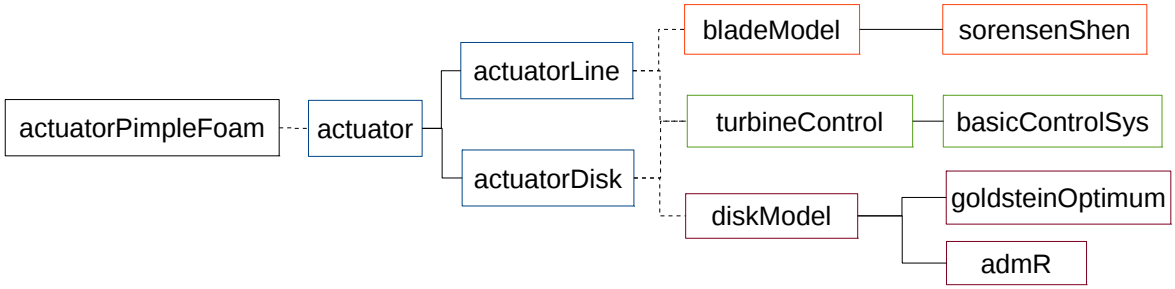

Figure 2: Diagram of the code outline with coloured class groups and showing lines of inheritance (solid) and dependence (dotted).

The dependence of the classes is managed via a member pointer to the class. This is a two way dependency whereby the upper level class has a member pointer to each of the instances of a lower level class and the lower level class has a member pointer to the upper level class. By handling the dependency in this way, both classes can access public member functions of each other. A direct inheritance line would not work for such a dependency, as the dependence is to a specific instance of the class and not simply requiring use of the class functions. Due to the cyclic dependence and dynamic allocation of the classes a pointer to the other class had to be used rather than a reference. The upper level classes control the creation and operation of the lower level classes and pass down the properties required for the operations.

4.2.2. Application Code

The application file reads in an input dictionary called “actuatorProperties” from the constant directory. The actuatorProperties file has a line declaring how many actuators are to be used and then includes sub-dictionaries for each actuator labelled “actuator_n” where n is the actuator number, starting at 0. A standard vector of pointers to the actuator base class is used as the container for the actuators; a for loop iterates for the number of actuators adding to the container for each model. The actuator model is found from the input in the actuator sub-dictionary and an if statement checks which type of model it is so that a pointer to the correct child class is used. Due to the templated nature of the standard vector it can hold pointers to different types of classes, provided they are derived from the same base class, therefore allowing combinations of different actuator models. The models are added using the

push_back function of the standard vector, which adds an element to the end of the vector. The keyword “new” is used to create a pointer to the new instance of the class and the constructor of the class has been written so that the actuator is ready for use immediately after construction.

The application uses a PIMPLE loop in the same way as pimpleFoam. A boolean tag is used to ensure that the actuators and volume force are only updated once per time-step, as this will increase efficiency. The actuators are updated in the UEqn file where they first clear the force from the points in the actuator disk region; the ALM has a disk-shaped swept area therefore a disk of points which may have a force on them. The volumeForce field that is used is the same from the start of the simulation and it is not replaced during the simulation as this is most efficient. Rather than sweeping the whole volumeForce field, only the points in the disk regions are cleared as again this is far more efficient. Each actuator is then looped over and adds the volume force that it calculated to the volumeForce field. The volumeForce field is dimensionless; this was decided upon since the elements of a dimensioned field lose their dimensions on access. Therefore a dimensionless force would need to be returned by the actuators in order to assign to the volumeForce elements. Since the volumeForce is merely a container for these dimensionless elements, whether or not it has dimensions becomes a null point, as it is merely used as a source term in the momentum predictor equation and at no other time. Giving volumeForce dimensions would require the user to insert the correct dimensions in the input files, but the dimensions are constant and not able to be adjusted by the user. Therefore it is far better from a user standpoint to leave it as dimensionless and to instead multiply it by a dimension converter term in the momentum predictor equation.

When writing the results the application loops over the actuators and accesses the total torque on them and their angular velocity. These are then written to csv files as with the SRF code in section 4.1.4. The actuator classes have been written so that the application only requires them to have four functions as: getPointsInDisk – returns the list of cells in the disk region of the actuator; updateAndAddToVolumeForce – updates the models, such as rotating the blades or recalculating the forces, and adds the new force to the field passed to the function; getTotalTorqueOnTurbine and getAngularVelocity – returns the torque and angular velocity respectively. Because the classes are only required to have these four functions, and only getPointsInDisk and updateAndAddToVolumeForce are critical to operation, additional sibling classes can be easily written to expand the modelling capability.

4.2.3. Actuator Classes

The base actuator class is abstract and has four pure virtual functions, the same four functions required by the application. From this class the actuatorDisk and actuatorLine classes are derived. Currently the base class does little more than allow for the polymorphic structure, as the functions that are common between actuatorDisk and actuatorLine are duplicated. However this will be changed before the code is released so that the common functions are defined in the base class. The actuator classes have a number of “set” and “get” functions, this method protects the member attributes from accidental modification as they can only be changed by “set” functions or a copy returned by the “get” functions. The “set” functions are included to allow the control system to fully manipulate the actuator object, these functions are: setActuatorSize and setActuatorPosition which each take in a standard map of the properties and the values to assign the hub and turbine radii or the centreline start and end points respectively. The functions setAngularVelocity and setAngularAcceleration allow the control system to control by the acceration, to replicate braking, or by the velocity itself. A further function, setAxialVelocity is included due to the different ways in which the local velocity sampling could be done. Were inlet sampling to be used, this would be done at the application level and would set the axial velocity known by the actuators using this function. Although local velocity sampling is not currently implemented, the structure exists to support it.

The common “get” functions are: getPointsInDisk, getAngularVelocity, getAngularAcceleration, getAxialVelocity, getInputDict, getTotalTorqueOnTurbine, getSizePropertiesMap and getPosistionPropertiesMap. The actuatorLine class has two further “get” functions, getUpDirection and getAllBlades. The function getUpDirection returns the reference point of the blades, since the actuators are fully defined by vectors and have no requirements for any orientations, an “up” direction can either be provided by the user or the blades will calculate one. The reason for referring to this as “up” is to provide some architecture for coupling to an ABL solver, which would require this for Coriolis force calculations. The getAllBlades function returns the blade pointers, allowing for manipulation by a control system.

When one of the child classes is created the constructor is called, for both child classes this takes in the sub-dictionary by reference as well as a pointer to the mesh and runTime objects. The constructor then assigns the dictionary and pointers to member attributes. The readProperties function is then called which reads through the dictionary and assigns values

to the rest of its required attributes. The setupTurbineControls function is then called; this checks the tag read in from the dictionary and redirects the turbineControlSystem pointer from the base turbineControls class to one of its child classes. Currently, this is only implemented for the actuatorLine class but the architecture is set-up for the actuatorDisk class.

The constructor then calls the findPointsInDisk function, this is common for both the actuatorLine and actuatorDisk classes due to the disk-shaped swept area of the ALM. The cells in the disk are stored by a standard vector container of integer labels to them. The function first clears the standard vector before iterating over the whole mesh and calling the pointIsInDisk function on the coordinates of each cell centre, accessed via the C member function of the mesh. If a point is found in the disk the push_back function of the standard vector is used to add the point to the container – if no points are found a warning is displayed. The pointIsInDisk function, shown below, takes in a Foam::vector as the input point coordinates. The radial distance of the point from the centreline and the axial distance along the centreline from the start point are both found and a boolean is returned of whether the distances imply the point is within the disk region.

```
bool Foam::actuatorDisk::pointIsInDisk(vector meshPoint)
{
    scalar radialDistanceOfPoint =
    (
        mag
        (
            (meshPoint - centreLineStartPoint)
          ^ (meshPoint - centreLineEndPoint)
        )
       /mag(centreLineEndPoint - centreLineStartPoint)
    );

    scalar axialDistanceFromDiskFace =
    (
        (meshPoint - centreLineStartPoint)
      & centreLineUnitVector
    );

    return
    (
        axialDistanceFromDiskFace >= 0
     && axialDistanceFromDiskFace <= diskThickness
     && radialDistanceOfPoint <= turbineRadius
     && radialDistanceOfPoint >= hubRadius
    );
}
```

The points in the disk are stored rather than recalculating them each time they are required. Assuming the actuator includes 2% of the cells in the mesh, the memory cost of storing these points is approximately 0.02MxA bytes, where M is the number of cells in the

mesh and A is the number of actuators, this would therefore be in the order of 10-100kb/A. For a full farm model this would be in the order of 0.1-10Mb, which is negligible in comparison to the cost of storing the mesh. The alternative is to iterate through the whole mesh and evaluate pointIsInDisk for each cell every time volumeForce is required. The time cost of this would be proportional to MxAxT, where T is the time cost for an evaluation of pointIsInDisk, for a full farm model this would be in the order of 10^9Ts, which would be around 10^1-10^3s per time-step. Thus significant time savings are made by storing the points in the disk.

The constructors of the actuatorDisk and actuatorLine models then create the diskModel and bladeModel respectively, in the same way as the turbineControls are created. For the actuatorDisk class the updateAndAddToVolumeForce function that is called by the application simply loops over the points in the disk and calls the calcVolumeForce member function of the diskModel class on each, assigning it to the respective point in the volumeForce field. When the actuatorLine class calls updateAndAddToVolumeForce it first updates the blades, to recalculate their force on the fluid, before looping over each and adding the blade volume force to the volumeForce field. The rotation angle is then calculated, which calculates the total torque by adding together the torque on each blade. If a free-spinning simulation is being performed then the angular acceleration and angular velocity are calculated as with equations 3 and 4 in section 3.1. The control system member functions controlAcceleration and controlVelocity are also called at this point. The rotation angle is then calculated as the angular velocity multiplied by the time-step. The rotateAllBlades function then loops over each blade and calls the bladeModel member function rotateBlade.

4.2.4. Disk Model Classes

The diskModel base class has only two pure virtual functions, calcVolumeForce and calcTotalTorqueOnBlade. The calcTotalTorqueOnBlade function is only used by the admR child class and is included to enable free-spinning disk models to be implemented, the member function in the goldsteinOptimum class merely returns zero when called. However, only some of the key architecture for this is currently implemented and an additional function is required to update the angular velocity. This function uses an implementation of equations 16-18 to find the torque. The constructors for the diskModels use the pointer to the actuatorDisk class to get the actuator properties, thus ensuring that if a control system modifies the actuator then the model is also modified. The input dictionary of the actuator is also accessed from the pointer to lookup a sub-dictionary of the inputs specific to the ADM.

The goldsteinOptimum child class implements the Goldstein-Optimum ADM, see section 3.2.1, so the constructor also calculates the constants for this model. The calcVolumeForce for this class simply calculates the scaled radius of the point from the centreline and finds the corresponding force, which it returns. The actuatorDisk class then adds this to the volumeForce.

For the admR class the ADM-R model is implemented, this shares a large portion of its functionality with the sorensenShen bladeModel class. Along with the aforementioned functionality of the constructors, the constructor of the admR class calls the initialiseDataStorage function, which the sorensenShen class also has. This function converts the sub-dictionary of the blade properties to the standard map container. The standard map is templated and allows a lookup key of the distance along the blade to be used to find the standard vector of properties at that point. The standard C++ containers are more flexible and better documented than those implemented by OpenFOAM. However OpenFOAM's dictionary structure is very useful for reading in data. An equivalent of the standard map of C++ in OpenFOAM would require one list for the distances along the blade that have properties at them and a list of lists for the properties, the link between the two would be done by the indexing. Although this could be implemented without issue, the map offers greater flexibility, in terms of contents, and has more cohesive documentation on its functionality as well as offering a more verbose operation in the code. This conversion, shown below, is done by using an iterator to loop over the contents of the dictionary, which are accessed by the “toc” member function which returns a wordList. Each entry is converted from a word to a string by assignment, as word is derived from string, which is then converted to a double by the “strtod” function and added to the standard vector of alongBladeDistances. A list corresponding to the entry is then read in from the dictionary and copied across to the standard vector for the data. The values in the map dataStorageDictionary are then assigned, with the key as the latest entry to the alongBladeDistances, which refers to the values in dataVector.

```
wordList dictionayContents = inputDictionarySubDict.toc();
List<scalar> dataList;
std::vector<scalar> dataVector;
string wordConvert;

for
(
    wordList::iterator entryIter = dictionayContents.begin();
    entryIter != dictionayContents.end();
    entryIter++
)
```

```
{
    dataVector.clear();
    wordConvert = *entryIter;
    alongBladeDistances.push_back(strtod(wordConvert.c_str(),NULL));
    dataList = inputDictionarySubDict.lookup(*entryIter);
    dataVector.reserve(dataList.size());
    std::copy
    (
        dataList.begin(),
        dataList.end(),
        std::back_inserter(dataVector)
    );
    dataStorageDictionary[alongBladeDistances.back()] = dataVector;
}
```

The calcVolumeForce of the admR class finds the axial distance of the point along the centreline as a vector projection along the centreline of the vector from the centreline start point to the mesh point. The radial vector is then found as the vector from the point along the centreline, that intersects with the plane defined by the mesh point and the centreline, to the mesh point. The linearInterpolateDataDict member function is then used with the magnitude of the radial vector, to linearly interpolate each of the blade properties to find their value at the mesh point. The lift and drag forces are then calculated based on the interpolated values and the force vector is created. The direction of the lift force is found as the negative of the cross product of the centreline unit vector with the radial unit vector, whereas the drag force acts down the centreline. The local fluid angle is then calculated and the force vector is rotated by the local fluid angle as the lift and drag act relative to the fluid rather than the blade. The Gaussian smoothing function, equation 13, is then calculated and multiplied by the force to scale the force accordingly.

The linearInterpolateDataDict function operates by first calling the calcBladeDistanceBounds function to find the interval of the blade properties that the mesh point lies within. This is done by a for loop that sweeps along the distances that the properties are defined at and compares them to the radius of the mesh point from the centreline. This could be improved for speed by using a recursive binary tree structure, see section 4.2.7. The linearInterpolateDataDict function then uses the data storage map of the blade properties to look up the values at each end of the interval and to linearly interpolate for the properties at the mesh point.

4.2.5. Blade Model Classes

The blade model class has seven pure virtual functions, four of these are for access namely: getBladeVector, which returns the vector defining the blade that can be set using the setBladeVector function; getPointsInBlade, which returns a standard vector of labels to the

mesh points in the blade; getBladeForces, which returns a standard vector of the corresponding volume force at each point in the blade and getTotalTorqueOnBlade, which returns the torque on the blade. The bladeModel base class also has a virtual class, calcTotalTorqueOnBlade, this is not pure virtual function and so has the definition as well. This function has an implementation in the same way as the diskModel child classes, in that equations 16-18 are used, however this is defined in the parent class and is fully operational.

The only child of the blade model class that has been written is the sorensenShen class. For this class the constructor reads in the necessary properties from the actuatorLine instance that creates it, including using the initialiseDataStorage equivalent to that used by the admR in section 4.2.4. In order to get the initial position of the blade the getUpDirection of the actuator is called and the perpendicular projection of this vector with the actuator centreline is checked. If it is zero, implying that it is parallel to the centreline, the function perpendicularUnitVector is called, this checks two mutually orthogonal vectors to find one with a perpendicular component to the centreline. The bladeVector is initially set as equal to the reference vector projection before the rotateBlade function is called to rotate it to the starting position.

In operation, the actuatorLine class first calls the update member function of the sorensenShen class, this function reads in the axial velocity from the actuator before calling the findPointsInBladeAndForces function. This function loops over the points in the disk that the actuator has already found. For each point the calcBladeGeometryAtPoint function is called, this calculates the vector from the centreline mid-point to the mesh point and projects this onto the blade to give the bladeLinePosition. The radius of the point to the blade is then found as the vector from the nearest point on the blade to the mesh point, as follows:

```
bladeLinePosition =
        (meshPoint - centreLineMidPoint) & bladeUnitVector;

radiusToBlade =
    mag
    (
        meshPoint
      - (centreLineMidPoint + bladeLinePosition * bladeUnitVector)
    );
```

The findPointsInBladeAndForces then calls the pointIsInBlade functions, which uses the previously calculated variables in a comparison with the bounds of the blade. In particular it checks that the bladeLinePosition is between the hub and turbine radii and the radiusToBlade is less than the maximum blade radius. If the point is found within the blade

the calcBladeForceAtPoint function is called. This uses the linearInterpolateDataDict function, equivalent to that of the admR in section 4.2.4, to find the blade properties at the point, before the local fluid angle is calculated along with the lift and drag forces. The lift and drag forces are then multiplied by unit vectors of the negative cross product of the centreline and the blade vector and the centreline respectively. These are then added together to get the force vector which is rotated about the blade vector using Rodrigues formula, equation 19, and multiplied by the Gaussian smoothing function, equation 14.

Following this the findPointsInBladeAndForces then adds a label to the mesh point to the standard vector container of the points in the blade and adds the force vector to a standard vector container of the forces corresponding to the points. After the findPointsInBladeAndForces function, the update function calls the calcTotalTorqueOnBlade function to update the total torque on the blade. This is a protected function, so the actuator cannot call it, this ensures that it is called at the correct time by the blade, instead the actuator can access it via the "get" function. After the actuator calls the update function it adds the forces to the volumeForce field and then calculates the rotation angle, based on the torque calculated by the blade. The actuator then calls the rotateBlade function which uses Rodrigues formula, equation 19, to rotate the blade vector about the centreline by the angle calculated by the actuator.

4.2.6. Turbine Control Classes

The turbineControl base class has only two pure virtual functions, controlVelocity and controlAcceleration. The class has a member pointer to the actuatorLine class, this could be changed to be a pointer to the base actuator class, therefore allowing it to function with actuatorDisk models as well. By having a pointer to the actuator that creates it, the control system class is able to access all the public functions of the actuator. The actuators were therefore written with get and set functions, as mentioned in section 4.2.3; these functions enable the control system to check and alter the properties of the actuators. They can therefore check if the acceleration or the velocity is too great and apply controls to them, hence the two functions already implemented. However the architecture exists to extend this to enable the pitch and yaw of the actuator to be adjusted and, with some further modifications, control of the angle of the blades and the coning angle of the turbine could be introduced. Since actuators can be used for a variety of applications, not just wind turbine modelling, the current architecture can allow additional code to cause the actuators to translate around the domain and the radii of them to be changed.

The turbineControl class has only one child class currently implemented, the basicControlSys class. This class is only capable of controlling the angular velocity and acceleration. The constructor of this class reads in the properties from a sub-dictionary, from within the actuator dictionary, that it is given by reference by the actuator. It can also access the actuator input dictionary from the get function to find out relevant constants and properties, such as the moment of inertia. The controlVelocity function checks whether the velocity is greater than the maximum specified value, if so, it sets it equal to the maximum. This function is not very realistic in terms of behaviour but allows an absolute limit of the velocity to be set. The controlAcceleration function operates by first checking if the power is greater than the maximum specified, if so, it will work out the difference in power required and find the change in angular acceleration required to bring the turbine down to the desired maximum power, assuming a constant torque. It then checks if the velocity is greater than the maximum specified and adjusts the angular acceleration accordingly. Finally it checks if the angular acceleration is greater than the maximum specified, in which case it is adjusted down to the maximum. This is achieved through the following code:

```
scalar angularAcceleration = parentActuator->getAngularAcceleration();

if
(
    parentActuator->getTotalTorqueOnTurbine()
  * parentActuator->getAngularVelocity()
  > maximumPower
)
{
    angularAcceleration =
    (
        parentActuator->getTotalTorqueOnTurbine()
      * parentActuator->getAngularVelocity()
      - maximumPower
    )
   /(parentActuator->getTotalTorqueOnTurbine()*runTime->deltaTValue());
}

if
(
    parentActuator->getAngularVelocity()
  + angularAcceleration*runTime->deltaTValue()
  > maximumAngularVelocity
)
{
    angularAcceleration =
    (
        parentActuator->getAngularVelocity()
      - maximumAngularVelocity
    )
   /runTime->deltaTValue();
}

if (mag(angularAcceleration) > maximumAngularAcceleration)
{
```

```
    angularAcceleration =
    maximumAngularAcceleration
  * (
        angularAcceleration
       /mag(angularAcceleration)
    );
}

parentActuator->setAngularAcceleration(angularAcceleration);
```

The controlVelocity and controlAcceleration functions are called by the actuator during the calcRotationAngle function, as this is the point whereby the torque has been updated but the blades have not.

4.2.7. Actuator Package Code Discussion

The actuator code is currently in a state whereby the code is fully functional and the architecture is very flexible. However there is a large amount of code inefficiency in that the code is often repeated in child classes rather than being defined in the base class. This often occurs in projects of this kind, as code duplication is often the fastest way to reimplement a method, however this is easily remedied by code reorganisation, which will be done before releasing the code. This will include creating a utilities class or code file which can be included in order to collect together the code which is used by most of the classes such as the linear interpolation functions, this would dramatically reduce the amount of code duplication. The lookup of properties along the blade is also currently inefficient as it uses a basic sweeping technique. This could easily be changed to a recursive binary tree search which is very code and time efficient. This method uses a bisection of the remaining points to be compared with, so that each comparison halves the number of remaining points. This method will also be implemented before release of the code.

The current control system is also very basic despite the architecture allowing considerably more complex systems; however, another group member has been investigating wind turbine control systems and so a more advanced one will be implemented in collaboration with them before release of the code. The velocity sampling is currently based on an input value in the dictionary, although fast, this method is very restrictive. This will be changed to allow inlet velocity sampling and localised velocity sampling before the code is released, this has also been investigated by another group member. The actuator code is lacking in some areas however these can be easily rectified and the exact methods as to how these fixes will occur have already been identified. The code does however have the architecture and underlying functionality for which it was designed; it is therefore successful

as the remaining issues are minor ones and do not effect the primary operation but merely the capability. Furthermore additional extensions to the code have been determined and will be investigated, these are detailed in section 5.2.

4.3. Test case results

4.3.1. Free-spinning SRF

The free-spinning SRF was found to behave as expected for the basic test case; contours showing the relative velocity magnitude and the case set-up are shown in Figure 3. The case was run for 20 seconds for a moment of inertia of $5x10^{-7}$ kgm^2 and $5x10^{-8}$ kgm^2. The angular velocity as a function of time for these cases is shown in Figure 4. The rotor with a lower moment of inertia initially oscillates and increases rapidly before suddenly decreasing to zero and continuing to oscillate. Whereas the high moment of inertia rotor has smaller oscillations and a more gradual change in angular velocity, although it too increases in angular velocity whilst maintaining the overall trend of slowing down. The oscillations are believed to have been initially caused by vortex shedding around the rotor. After the initial oscillations are created, the angle of the rotor blades to the flow change the forces on the rotor thus continuing the oscillations. Once the rotor reaches a certain angle, the blade which was perpendicular to the flow has a lower restoring force than the advancing force caused by the blade which had been parallel to the flow. At this point the rotor receives positive feedback increasing its velocity; the same effect can happen in reverse to give negative feedback. It is thought that this process drives the larger oscillations and causes the more dramatic changes in angular velocity.

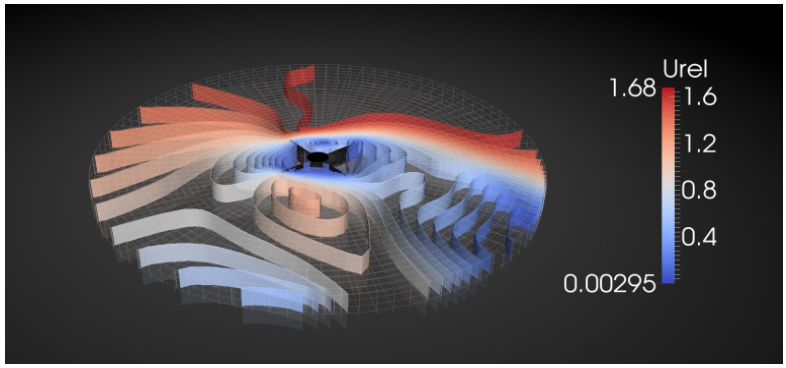

Figure 3: Relative velocity contours for the vertical rotor in a cross flow test case

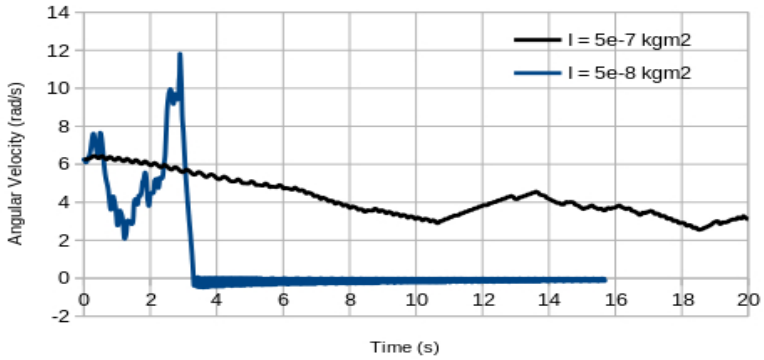

Figure 4: Angular velocity as a function of time for moments of inertia of $5x10^{-7}$ kgm^2 and $5x10^{-8}$ kgm^2 for the vertical rotor in a cross flow test case.

Although the basic test case demonstrates the free-spinning aspects of the SRF code, it does not represent a wind turbine and so a second test case was run to test the viability of the use of a free-spinning SRF on a wind turbine geometry. This test was found to be very computationally expensive. The initialisation was done first with a potentialFoam run to get an approximate initial velocity and fluxes, from this a SRFSimpleFoam case was run for 2000 iterations to get a steady-state approximation. At this stage the results were showing solver instability around the outlet. The case was therefore run for 2 seconds of SRFPimpleFoam in an attempt to flush the instability out of the domain however this again failed – despite an adjustable time-step maintaining a CFL number of 0.7. Various combinations of initialisation were attempted including using SRFPimpleFoam from an uninitialised domain however none of these attempts were successful. As initialisation failed, the free-spinning aspects were unable to be tested fully but they were run from an uninitialised domain however this also failed. It was noted that, in order to achieve 2 seconds of simulated time, the simulation was required to run for approximately 2 weeks, for a mesh of around 1.5 million cells; a relatively cheap mesh for a simulation of this type. This level of computational expense is not viable for use in free-spinning investigations which, as demonstrated, require a long period of simulated time – in the order of tens to hundreds of seconds, depending of the moment of inertia.

4.3.2. Actuator package

The tests on the actuator package had two main purposes, one to test the free-spinning aspects of the ALM and the second to test the code functionality. The former being the most detailed as the exact behaviour was under scrutiny rather than the latter which was simply a pass-fail manner of testing in order to find bugs in the code. In order for the free-spinning

turbine to be successful in the tests it must spin up to speed and slow down. The spinning up to speed should see one case whereby the control system limits the velocity and another whereby the turbine accelerates to its theoretical maximum where the torque produced eventually drops off as it is countered by the drag force; this is shown in Figure 5. It is clear that this does occur, therefore suggesting that the acceleration aspects of the free-spinning ALM are operational. The control systems are limiting the velocity by applying a counter-torque to cut the acceleration therefore this creates a fluctuating response, the amplitude of which varies with wind-speed. The implemented control systems can also therefore be deemed to be operational as well. The higher wind-speed also creates a faster acceleration and a higher maximum velocity, which is expected, adding further credence to the functionality of the free-spinning ALM.

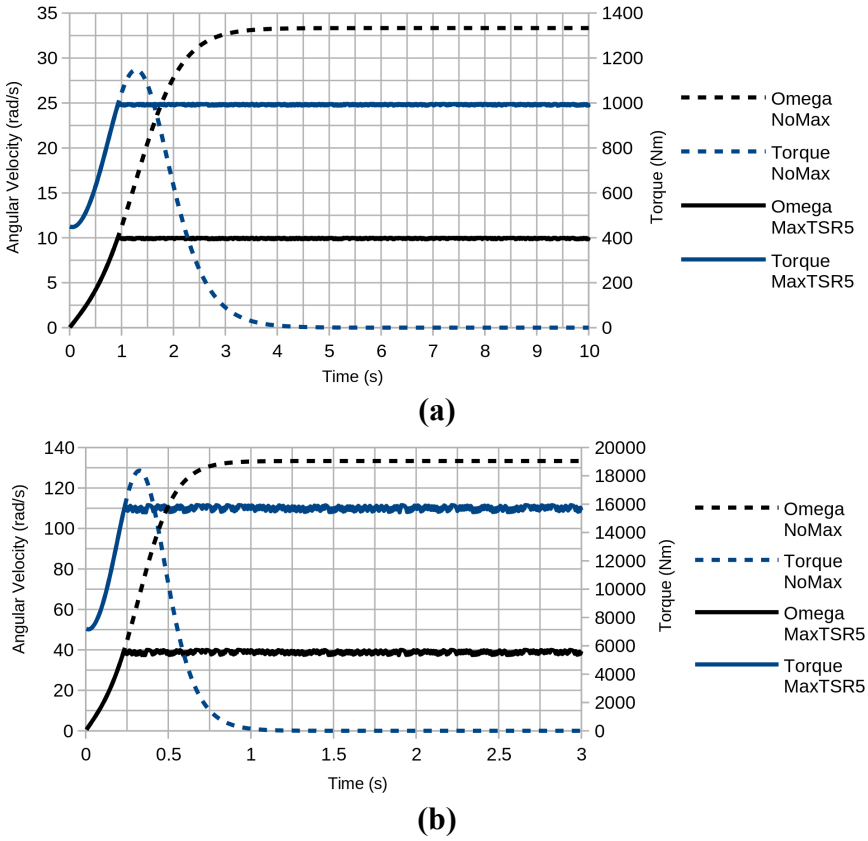

Figure 5: Angular velocity and torque as a function of time for a wind speed of 5m/s (a) and 20m/s (b) for the spin-up test cases. For cases with control systems limiting the angular velocity to a TSR of 5 (solid) and without an imposed limit (dotted).

The other aspect of the free-spinning ALM that is required to be tested is the deceleration; this is shown in Figure 6. It is expected that the turbine will approach the velocity that it is slowing down to, however it will not reach it entirely as the torque slowing the turbine will decrease as the velocity tends to the natural velocity. For the still air case, this

will mean the turbine tends towards zero velocity and the torque will also tend towards zero. This case is not realistic in that the turbine would ordinarily have friction torque from the generator and bearings which will add to the torque and cause it to stop entirely. Although this torque can be added in in the actuator package, it is important to test the actuator behaviour without it to ensure that the drag forces are operating correctly. It is clear that the deceleration of the free-spinning ALM is also operating as required. The deceleration test is vital for ensuring the turbine does not have positive feedback. As the geometry and forces are all defined by vectors, if one is negative when it should not be then the forces add positive feedback and the turbine cannot decelerate. However this cannot be detected simply from an acceleration test, unless the test is run to find the theoretical maximum velocity. Therefore both acceleration and deceleration tests are required.

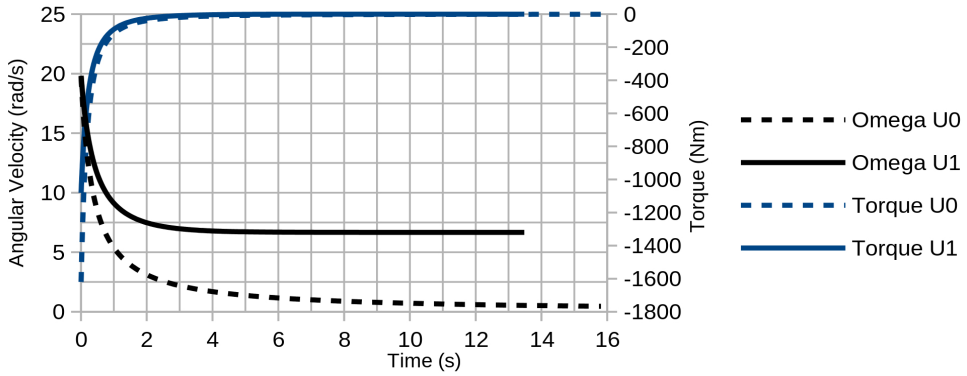

Figure 6: Angular velocity and torque as a function of time for the deceleration test cases in still air (dotted) and in a 1m/s wind (solid)

The other actuator package test cases were performed to ensure that the implemented features and functionality operated correctly. From these tests it was found that the code had very few bugs at all, those that did exist were addressed without issue. Therefore these cases showed that code can run any number of turbines, with any combination of control systems on those turbines. The ALMs could have different numbers of blades and can be run alongside ADMs. The actuators can have different spin speeds and orientations, including reverse spinning. It was found from wake-visualisation using Q-criterion that the wake was far away from the domain boundaries without the domain being of excessive size. Therefore the test domain size is suitable for further investigations, however a mesh sensitivity study should be performed alongside model validation in order to give a recommended set-up.

5. Conclusion and Future Work Recommendations

5.1. Conclusion

The free-spinning SRF code was implemented into the existing SRF code in OpenFOAM. The modification was efficient and proved to be successful in the basic test case, however in the more advanced test case the SRF became unstable and very computationally expensive. It is therefore believed to be unviable as a method of investigating free-spinning behaviour due to this instability and expense and the long simulated times required by free-spinning investigations. It is particularly unviable in the field of wind-turbine predictions because a further drawback is that it can only simulate a single turbine and not a farm due to the underlying methodology.

The actuator package, which was written in OpenFOAM, was successful in the code implementation and was shown to be very flexible and highly efficient. The free-spinning ALM was successful on the acceleration and deceleration tests and only requires the implementation of local velocity sampling to perform more complex investigations. The ability to run multiple turbines with different models and control systems in any orientation and position allows full farm models to be constructed without difficulty. The polymorphic structure underlying the actuator package allows any models and control systems to be written into the package, without the need to rewrite large parts of the code. Therefore with only a few additional features, the actuator package will serve as a platform for all types of investigations into the discrepancies between free-spinning and prescribed velocity wind-turbine models. It will also enable actuator models to be easily added to the package to test them for their accuracy. Thus the actuator package will remove a large number of the obstacles that may be faced in researching actuator or free-spinning simulation techniques. Consequently allowing a further investigation to better identify the effects of wake-shadowing, wind fluctuations and the theoretical maximum angular velocity of turbines than can be achieved through prescribed velocity simulations.

5.2.Future Work

5.2.1. Hub Modelling

In ALMs the turbine hub is often left empty and applies no force on the fluid. This has the effect of causing the turbine to act as a funnel at the hub, accelerating flow through the centre as there is no opposing force at this region. Clearly, in reality the hub would apply a drag force preventing this effect. The NREL code applies a hub model to slow the air through

the centre of the turbine, this can be achieved through the use of a specialised ADM. An additional feature for the actuator package will be to add in this hub-modelling.

5.2.2. Local Velocity Sampling

In order to give full functionality to the free-spinning ALM, local velocity sampling must be implemented. This will be done to allow the turbines to respond to either the inlet velocity, a disk sample region in front of the turbine or, for the ALM, individual blade sample regions in front of each blade. These will either calculate an average for the regions or will create a database of values, from which the velocities corresponding to each cell in the volume force regions can be found.

5.2.3. Complex Control Systems

The control systems of wind-turbines can be very advanced, with changes to pitch, yaw, coning angle, blade angle and angular velocity all possible. The code architecture allows the control system to change these factors but currently there is no implemented control system for this. Therefore one additional task will be to create such a control system. The abilities of the control system implemented in the actuator package also enables turbine orientation and position changes to be made, therefore an additional control system may be written which is capable of moving the turbines. An example of this application is in floating off-shore wind farms, where the platforms can pitch and roll.

5.2.4. Free-spinning actuator disk

The ADMs implemented in the actuator package have the ability to be made free-spinning. The Goldstein-Optimum model would require additional data about the turbine for this, however the ADM-R would have a very similar operation to the free-spinning ALM already implemented. The reason for creating a free-spinning ADM will be for a cheap modelling alternative when investigating the far-wake effects. This could easily be implemented in the actuator code package.

5.2.5. Templated Meta-Programming

In C++ an advanced coding technique, which is an extension beyond polymorphism, is templated meta-programming. This allows functions and operations to be called on any type of class, provided they are called in the same manner. The more recent versions of OpenFOAM have used an underlying templated structure to modify solvers, such as by adding in MRFs to specific cell regions. Consequently, this would allow the actuator package to be used in any solver rather than limiting it to a specific application. This would have

particular use in tidal-turbine simulations where free surface modelling solvers are used. Implementing the actuator package into this templated structure will be investigated.

5.2.6. ABL Solver

An ABL solver will be combined with the actuator package to enable atmospheric effects to be included in the simulation. Another group member investigated combining an ABL solver with an ADM and identified the ABL as having a large influence on turbine performance. Therefore this would be done in collaboration with them.

5.2.7. Validation

The actuator code has currently only been tested for functionality and the implementation of the models have not been validated. In order for the accuracy of the code to be checked and for the code results to be reliable, validation cases must be run to compare the results to experimental and DNS results. Therefore this will also be done to accompany the release of the code.

6. Project management, consideration of sustainability and health and safety

The project was managed by a Gantt chart which set out which tasks were dependent on others and the time each task should be taking. Coupled to this a risk assessment was performed at the start of the project to identify which risks were most likely to develop, so their progress could be monitored and action could be taken to mitigate their effects. It was by this method that the risk of the SRF becoming unviable, due to a long run-time and potential instability, was identified as the primary risk to the project. It became apparent when the complex test cases had begun that the cases would take a long time; therefore whilst the test cases were running the development of the actuator package was started to mitigate the potential risk. After all the possible methods for stabilising the SRF had been exhausted, the next step for the SRF development was to attempt a simpler axial case. This had the issue in that it was very labour-intensive and the risk of failure was identified as very high with little gain; consequently the decision was made that the SRF had become too unviable for the project and that further development of it should be stopped to instead focus on the actuator code. It was by this decision making, based on the information given by the Gantt chart and identified in the risk assessment, that the unviability of the free-spinning SRF had very little effect on the overall success of the project and the risk it introduced was contained.

There were no major health and safety risks associated with the project. However the risks associated with long periods of time at a computer were mitigated by ensuring that the desk, chair, keyboard and screen were all positioned to reduce glare and acute joint angles. The sustainability aspects of the project are also minimal, however it is worth considering that, as high performance computers (HPCs) are very energy intensive, code efficiency improvements that can reduce the computation time have a major impact on the energy use.

7. Contribution to group functioning

This project was part of a larger group project investigating the discrepancies between predicted and realised performance of wind-turbines; the group project is run alongside the larger industry project, SWEPT2, which has similar aims. It was identified that currently all wind-turbine simulations are based on averaged data and prescribed angular velocities; however there are also no existing solvers capable of running free-spinning simulations. Therefore, in order to investigate the discrepancies between free and prescribed angular velocity simulations, a free-spinning solver was first required to be written. This would then allow a later project to investigate the additional accuracy that a free-spinning solver enables.

This project links in with a number of the other group members primarily through the actuator package. One group member, Louis Hudson, was investigating turbine control systems; this linked into the actuator package through influencing the manner of the implementation of the control systems. This will have further links as a more advanced control system will be implemented before the code is released. Another member, Ben Ashby, investigated local velocity sampling for actuators which will also be implemented in the actuator package before release. George Hyde-Linaker investigated combining an ABL solver with an ADM; this method will be included in the actuator package as it was found that an ABL has a large effect on the performance of the turbines. Furthermore, assistance was given to other group members, in particular, when advice on coding or OpenFOAM was required. It was necessary to support Ben Ashby with implementing the local velocity sampling, due to a greater experience in C++ coding and OpenFOAM's structure. Assistance was also given to Matthew Howard at the start of the project to explain Svenning's [13] implementation of the ADM and how to modify the code. Support was also given to George Hyde-Linaker in advising on how to combine the ABL method with the actuator disk.

References

[1] Li, Y. Castro, A. M. Sinokrot, T. Prescott, W. and Carrica, P. M. Coupled multi-body dynamics and CFD for wind turbine simulation including explicit wind turbulence. *Renewable Energy.* 2015, **76**, pp 338-361.

[2] Li, Y. Paik, K. Xing, T. and Carrica, P. M. Dynamic overset CFD simulations of wind turbine aerodynamics. *Renewable Energy*. 2012, **37**, pp 285-298.

[3] Bazilevs, Y. Hsu, M. C. Akkerman, I. Wright, S. Takizawa, K. Henicke, B. Spielman, T. and Tezduyar, T. E. 3D simulation of wind turbine rotors at full scale. Part I: Geometry modelling and aerodynamics. International Journal for Numerical Methods in Fluids. 2010, **65**(1-3), pp 207-235.

[4] Tran, T. and Kim, D. The platform pitching motion of floating offshore wind turbines: A preliminary unsteady aerodynamic analysis. *Journal of Wind Engineering and Industrial Aerodynamics*. 2015, **142**, pp 65-81.

[5] Sezer-Uzol, N. and Long, L. N. 3D Time-accurate CFD simulations of wind turbine rotor flow fields. *American Institute of Aeronautics and Astronautics (AIAA) Aerospace Sciences Meeting, Jan 2006, Reno NV*. Reston VA: AIAA, 2006, pp 1-23.

[6] Sanderse, B. van der Pijl, S. P. and Koren, B. Review of computational fluid dynamics for wind turbine wake aerodynamics. *Wind Energy*. 2011, **14**, pp 799-819.

[7] Sørensen, J. N. and Shen, W. Z. Numerical Modelling of Wind Turbine Wakes. *Journal of Fluids Engineering.* 2002, **124**(2), pp 393-399.

[8] Martinez-Tossas, L. A. and Leonardi, S. Wind turbine modelling for computational fluid dynamics. Golden CO: National Renewable Energy Laboratory (NREL), 2013.

[9] Churchfield, M. J. *OpenFOAM Workshop Training Session: Wind Energy and Atmospheric Boundary Layer Tools and Tutorials*. 6Th OpenFOAM Workshop*,* Pennsylvania State University, Pennsylvania, 13-16 June 2011.

[10] Peet, Y. Fischer, P. Conzelmann, G. and Kotamarthi, V. Actuator line aerodynamics model with spectral elements. *51St AIAA Aerospace sciences meeting, Jan 2013, Grapevine TX.* Reston: AIAA, 2013, pp 1-7.

[11] Mikkelsen, R. *Actuator disc methods applied to wind turbines*. Ph.D. thesis, Technical University of Denmark, 2003.

[12] Nodeland, A. M. *Wake modelling using an actuator disk model in OpenFOAM.* Master thesis, Norwegian University of Science and Technology, 2013.

[13] Svenning, E. *Implementation of an actuator disk in OpenFOAM*. CFD with open-source software. Chalmers University of Technology, 2010.

[14] Jeromin, A. Bentamy, A. and Schaffarczyk, A. P. Actuator disk modelling of the Mexico rotor with OpenFOAM. *First symposium on OpenFOAM in wind energy, March 2013, Oldenburg.* London: EDP sciences, 2014, pp 1-12.

[15] Goldstein, H. *Classical Mechanics*. 3rd ed. Reading MA: Addison-Wesley, 1980. p 162.