



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Escola Superior d'Enginyeries Industrial,  
Aeroespacial i Audiovisual de Terrassa

# CFD Simulation of a Floating Wind Turbine in OpenFOAM: an FSI approach based on the actuator line and relaxation zone methods.

## Document:

Appendix

## Author:

Pere Frontera Pericàs

## Director:

Daniel Garcia-Almiñana

## Degree:

Master's degree in Aerospace Engineering

## Examination Session:

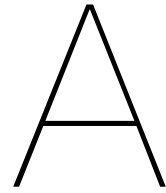
Autumn 2022

MASTER FINAL THESIS

# Contents

<b>A</b>	<b>Introduction to the OpenFOAM environment</b>	<b>1</b>
A.1	The environment . . . . .	1
A.2	Solvers . . . . .	1
A.2.1	The sequential approach . . . . .	1
A.2.2	Multiphase solvers . . . . .	2
A.3	Discretization . . . . .	2
A.4	Meshing . . . . .	3
A.5	Boundary conditions . . . . .	4
<b>B</b>	<b>Rigid body motions and loads</b>	<b>5</b>
B.1	Geometric definitions . . . . .	5
B.1.1	Frames of reference . . . . .	5
B.1.2	Rotation notation . . . . .	6
B.2	Implementation of new restraints . . . . .	8
B.2.1	Constant loads . . . . .	8
B.2.2	Gyroscopic moment . . . . .	8
<b>C</b>	<b>Scripts and templates</b>	<b>10</b>
C.1	waves2Foam . . . . .	10
C.2	dynamicMeshDict. . . . .	11
C.3	Harmonic turbine motion . . . . .	13
C.4	IOdictionary . . . . .	15
	<b>References</b>	<b>17</b>





# Introduction to the OpenFOAM environment

Even though OpenFOAM is one of the most popular tools for CFD simulations, the learning curve can be steep for newcomers, especially due to the lack of a user interface and an official manual. Therefore, this section presents a brief and very general introduction to the main aspects of CFD simulation in OpenFOAM.

## A.1. The environment

The *Open Source Field Operation and Manipulation* (OpenFOAM) is an open-source set of C++ libraries dedicated to creating *applications* of two types: *solvers* that are meant to numerically solve a set of differential equations from continuum mechanics, and *utilities* to perform specific tasks through data manipulation. OpenFOAM also includes pre- and post-processing environments. To define a specific simulation, the user has to define a *case* folder with the following subdirectories:

- **constant.** Contains the problem properties that will not change during the simulation, such as the mesh description in the subfolder `polyMesh` or the physical properties in the file `transportProperties`.
- **system.** Defines the solution procedure and contains at least three files: `controlDict` to set run control parameters, `fvSchemes` to specify the discretization schemes and `fvSolution` where solvers, tolerances and algorithm controls are defined.
- **Time directories.** Set of folders containing the output of the simulation at the specified time intervals. The name of each time directory is based on the simulation time at which the data is written. At least one folder containing the initial conditions has to be created.

Each file must follow a simple set of syntax rules. OpenFOAM uses *dictionaries* as a mean to specify different types of data. All files start with a dictionary called *FoamFile* containing a standard set of keywords that help OpenFOAM identify the file and its contents.

## A.2. Solvers

OpenFOAM solves the coupled pressure-momentum system from the Navier-Stokes equations using different techniques that depend on the actual case.

### A.2.1. The sequential approach

For the incompressible equations, momentum and mass transport equations are decoupled from the energy equation, which does not need to be solved to obtain the pressure  $\mathbf{p}$  and velocity  $\mathbf{u}$  fields. However, pressure terms are not present in the mass conservation equation, and thus special techniques are required to couple and solve the system. The main strategy in such a case is to apply the divergence operator to the momentum equation, discretize the time derivative terms, and then substitute

the mass conservation equation. This will lead to a Poisson-type equation for the pressure, resulting in a linear system of four equations with four unknowns: three velocity components and pressure. These equations are not solved directly but sequentially, and depending on the specific strategy, three main algorithms can be defined [1]:

**SIMPLE: Semi-Implicit-Method-Of-Pressure-Linked-Equations.** This method does not consider any time derivative and thus stays within a single time-interval, thus being reserved for steady-state solutions. Moreover, this solver is not consistent because the pressure term is only approximate and requires under-relaxation to ensure numerical stability. Smaller relaxation factors will result in greater stability but prolonged convergence rates. The consistent version of the algorithm (*SIMPLEC*) includes the missing pressure term, requiring more iterations for each calculated step but decreasing the convergence rate.

**PISO: Pressure-Implicit-of-Split-Operations.** Contrarily to *SIMPLE*, it includes time-derivative terms and consistently solves the system through one predictor and two corrector steps. Under-relaxation is no longer needed, but to ensure numerical stability, the CFL number must remain below 1 at each cell, and thus the time-step should be selected accordingly. The *PISO* solver is indicated for transient simulations.

**PIMPLE: Merged PISO-SIMPLE.** One of the most used solvers, it takes the advantages of the two previous ones in the sense that it can be applied to transient simulations with CFL numbers larger than one, allowing for larger time-steps. Within each time-step, the under-relaxation procedure from *SIMPLE* is applied and a “steady-state” solution is found. But before advancing to the next time step, outer-correction loops are performed to ensure that the explicit part of the equation has converged. If no outer-correction loops are applied, then the algorithm works in *PISO* mode.

Solver controls, tolerances, and algorithms are defined in `fvSolution`.

### A.2.2. Multiphase solvers

Apart from these three, many different solvers are natively implemented in OpenFOAM. Special attention will be given to `interDyMFoam`, a solver for two incompressible fluids using a VOF approach with support for moving meshes. Still, multiple options exist to solve the multiphase problem (equation 2.24):

**MULES.** The *Multidimensional Universal Limiter with Explicit Solution* is used in the `interFoam` solver along with the artificial interface compression mechanism explored in section 2.2.2. MULES’s task is to bound the volume fraction field  $\alpha \in [0, 1]$  by employing high order treatment at the interfacial region [2]. OpenFOAM’s implementation uses a semi-implicit variant of MULES (`MULEScorr`) which maintains boundedness and stability at arbitrarily large CFL numbers.

**isoAdvector.** This method, incorporated into the `interIsoFoam`, advocates for a purely geometrical formulation for interface advection and reconstruction. By taking sub-time steps, the flux of the indicator field  $\alpha$  is calculated analytically by assuming that the interface moves at a steady pace during the time-step. This method is consistently second-order in all mesh types [3].

Although initially thought to produce sharper interfaces and require fewer computational resources [4], the *isoAdvector* performance is very similar to MULES’ in the context of linear wave propagation across a wave tank [5]. For more complicated interfaces, such as rising and breaking bubbles, the *PLIC* reconstruction method proposed in [3] enhanced the quality of the results obtained with *isoAdvector*.

## A.3. Discretization

The different solvers solve the differential equations by mapping them into a discrete domain so that the continuous set of equations is transformed into a system of linear algebraic equations. To do so, the different differential operations must be discretized, i.e., approximated along the grid using the FVM approach. The user can define which numerical schemes to use for each mathematical term in the `fvSchemes` dictionary using the following keywords:

- **interpolationSchemes.** Point-to-point interpolations. Examples are *linear* and *midPoint*.

- **snGradSchemes.** Gradient component normal to a cell face. They can be corrected for surface non-orthogonal orientation.
- **gradSchemes.** Gradient ( $\nabla$ ). If a *Gauss* scheme is selected, then the user must specify which interpolation technique to use.
- **divSchemes.** Divergence ( $\nabla \cdot$ ). A *Gaussian* scheme is used for discretization and the user must specify the interpolation scheme.
- **laplacianSchemes.** Laplacian ( $\nabla^2$ ). A *Gaussian* scheme is used for discretization and the user must specify the interpolation and surface normal gradient schemes.
- **timeScheme.** First and second time derivatives. Examples are *Euler*, *backward* and *CrankNicolson*.

## A.4. Meshing

The mesh being a crucial part of the simulation, OpenFOAM has in-built utilities to define meshes with a very general structure (`polyMesh`) that will automatically meet the validity constraints. The latter is not guaranteed for meshes generated by third-party software, but bear in mind that mesh generation in OpenFOAM is not as straightforward and visual as with other tools. It is important that when defining cell faces in OpenFOAM, the face-normal obeys the right-hand rule. Through the `cellShape` tool, OpenFOAM can define a wide variety of cell shapes such as hexahedrons, wedges, and prisms. OpenFOAM allows for the automatic generation of meshes using two utilities:

**blockMesh.** The domain is divided into a set of hexahedral blocks with straight or curved edges. Again, the right-hand rule must be followed when defining the block vertices. The user can then specify, for each block, the number of cells in each direction (cell expansion ratios). The boundary faces (*patches*) must also be specified. For meshes with more than one block, these can be connected using face matching or face merging.

**snappyHexMesh.** The domain is divided into hexahedra and split-hexahedra from surface geometries in Stereolithography (STL) format by iteratively refining an initial mesh (typically generated using *blockMesh*) and then morphing it to the STL surface. The surface STL data is defined in the `triSurface` sub-directory, whereas the mesh control entries are given in the `snappyHexMeshDict` dictionary. The initial mesh should have an aspect ratio close to 1 near the STL surfaces. The refinement of the cell splitting around feature surfaces is controlled in `castellatedMeshControls`. Additional layers of hexahedral cells aligned to the boundary surface can be added with the `addLayersControls` sub-dictionary. Mesh quality is controlled in the `meshQualityControls` sub-dictionary.

For the cases of moving meshes, the controls regarding grid deformation and morphing are defined in the `dynamicMeshDict` dictionary. It allows four types of mesh motions, the most applicable to the current case being the mesh motion based on solving the dynamics of a rigid body (*dynamicMotionSolverFvMesh*). This will be extremely useful since it allows for coupling between the rigid body and the flow boundary conditions. In the case of deforming meshes, it also lets the user choose a diffusivity parameter that controls how the mesh motion is distributed over the domain. The user can also define inner and outer distances that determine whether the mesh is moving with the rigid body or does not move at all. Special attention has to be given to the mesh quality during the simulation since cells may be heavily deformed or shrunken, leading to a smaller CFL number and thus affecting the numerical stability.

Meshes with changing topological features require appropriate consideration. For the floater, a sliding mesh technique provides a good balance between robustness and complexity, but note that cell connectivities (i.e. mesh topology) will change during the simulation. In OpenFOAM, a spherical *slave* mesh attached to the floater can be defined and coupled to a fixed *master* mesh using an Arbitrary Mesh Interface (AMI) that projects one of the patches' geometry onto the other. To reduce interpolation errors, these meshes should be similar near the coupling patches. Although an overset mesh technique seems to be more adequate for the case of a floating body, it uses non-conservative interpolation, needs higher computation times, and requires careful planning and expertise.

## A.5. Boundary conditions

The FVM aims to solve the well-known boundary-value problem, which results from the combination of a set of partial differential equations with their corresponding boundary conditions (BCs). Thus, a correct definition of the latter is crucial for the success of any simulation. Ill-posed conditions can lead to unphysical solutions and even solver failure. For incompressible flow and single-phase inlets, exactly four variables must be specified as Dirichlet boundary conditions. On the other hand, outlets need exactly one variable as a Dirichlet condition. In OpenFOAM, more than 70 different types of BCs are available in three categories: `basic`, `constraint` (for geometric constraints) and `derived` (for specialized conditions). The latter includes specific BCs for `inlet`, `outlet` and `wall` patches. These conditions are specified in the `boundaryField` dictionary in the `0` subdirectory.

# B

## Rigid body motions and loads

This chapter is meant as a comprehensive summary of the different instruments used to describe the rigid-body motions, characteristic of FOWTs. Section B.1 presents the four frames of reference recurrent in the present research for the description of FOWT dynamics. Emphasis is also given to the mathematical notation required to describe rigid-body rotations in 3D space. Then, section B.2 moves the focus towards the implementation of new restraints (e.g., weight and gyroscopic coupling moment) for the simulation of rigid bodies in OpenFOAM.

### B.1. Geometric definitions

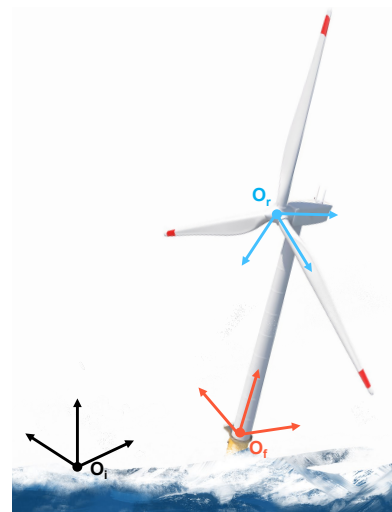
Given that an FOWT floater is commonly modeled as a rigid body, its motion can be described as the translation of its center of rotation plus a rotation around the latter point. This seemingly simple scenario can easily become cumbersome if not handled properly. This section aims to define the different frames of reference involved in the dynamics of FOWTs and present the geometric transformations required to characterize their motion.

#### B.1.1. Frames of reference

The simplest approach to modeling a FOWT is to consider the floater and rotor as different rigid bodies, thus moving relative to one another and, at the same time, to a fixed *inertial* frame of reference  $i$ . Being able to describe the body's motion in the latter frame is of great interest since it requires no fictitious forces to be taken into account. However, not all the quantities are easy to describe in the inertial frame, calling for the need for body-attached non-inertial frames: the floater  $f$ , rotor  $r$  and principal  $p$  frames. These four frames are defined as follows:

**Inertial frame  $i$ .** The choice of inertial frame is rather arbitrary, since all are equivalent upon Galilean transformations. In the present work, the chosen inertial frame corresponds to OpenFOAM's default frame of reference: the origin  $\mathbf{O}_i$  is kept fixed at  $(0, 0, 0)$  while the axes are parallel to  $XYZ$ .

**Floater frame  $f$ .** The origin of the floater frame  $\mathbf{O}_f$  corresponds to the instantaneous center of rotation of the floater and varies in time as the FOWT undergoes linear motion (surge, sway and heave). The frame axes are kept fixed with respect to the floater body, but rotate along with it. Consequently, the floater frame rotates with



**Figure B.1:** Frames of reference of a FOWT assuming a blade azimuth angle  $\Psi = 120^\circ$ .

respect to the inertial frame. Floater axes are chosen so that they are aligned with inertial's when the floater is undergoing no rotation (no roll, pitch nor yaw).

**Principal frame  $p$ .** The principal frame shares origin with the floater frame and is attached to it. The only difference with the floater frame is a constant shift in the axes orientation, which are aligned with the floater principal axes of inertia. This frame of reference is seldom used, but still it is useful for solving the rigid body equations and thus is used in OpenFOAM for this same purpose.

**Rotor frame  $r$ .** The origin of the floater frame  $\mathbf{O}_r$  corresponds to the instantaneous center of rotation of the rotor. For most cases, its position with respect to the floater frame is kept fixed. The third component of the rotor axes coincides with its rotation axis, while the other two are kept in the rotor plane.

The floater, inertial and rotor frames are depicted in Figure B.1. Now that all four frames are well described, it is easier to define the six DoFs characterizing the rigid body motion of the FOWT (refer to section 1.1 for the fundamentals on FOWTs):

- The linear degrees of freedom  $\mathbf{d} = \{\text{surge, sway, heave}\}^T = \{x, y, z\}^T$  are defined as the position of  $\mathbf{O}_f$ , given in the inertial frame.
- The angular degrees of freedom  $\boldsymbol{\phi} = \{\text{roll, pitch, yaw}\}^T = \{\phi, \theta, \psi\}^T$  are defined as the Tait–Bryan angles that describe the floater frame rotation from the inertial frame.

### B.1.2. Rotation notation

The linear motion of a body is straightforward to represent, given its position vector  $\mathbf{d}$  as a function of time. Rotations, on the other hand, are more challenging to encode. This section aims to clarify the mathematical notation used to describe rotations in Euclidean space and present two of the most widely used rotation representations.

#### Rotation transformations

In a nutshell, rotations can be thought of as linear transformations that rotate a vector  $\mathbf{v}$  around the origin. The result of this transformation is the vector after the desired rotation,  $\mathbf{v}'$ . The same can be applied to a second (or higher-order) tensor  $\underline{\mathbf{T}}$ . These transformations actually change the physical position or orientation of the point or body of interest, and thus are known as active transformations. They are encoded into what is known as rotation matrix  $\underline{\mathbf{R}}$ :

$$\mathbf{v}' = \underline{\mathbf{R}} \mathbf{v}, \quad \underline{\mathbf{T}}' = \underline{\mathbf{R}} \underline{\mathbf{T}} \underline{\mathbf{R}}^T \quad (\text{B.1})$$

Rotation matrices are orthogonal with  $|\underline{\mathbf{R}}| = 1$ , thus  $\underline{\mathbf{R}}^{-1} = \underline{\mathbf{R}}^T$ . Because all rotations are defined along the origin, rigid body rotations around the CoR must apply a coordinate shift before and after the transformation. For the FOWT case where  $\mathbf{v}$  represents a generic floater point (given in inertial frame) and the CoR coincides with  $\mathbf{O}_f$ , then the position after rotation is computed as:

$$\mathbf{v}' = \underline{\mathbf{R}} (\mathbf{v} - \mathbf{O}_f) + \mathbf{O}_f \quad (\text{B.2})$$

To describe the change in position from one time instant to the following one, then the linear displacement  $\Delta \mathbf{d}$  must be used to update the position of the CoR:

$$\mathbf{v}^{n+1} = \underline{\mathbf{R}} (\mathbf{v}^n - \mathbf{O}_f^n) + \mathbf{O}_f^{n+1}, \quad \mathbf{O}_f^{n+1} = \mathbf{O}_f^n + \Delta \mathbf{d} \quad (\text{B.3})$$

This approach has been used in Section 6.2 to update the position of the actuator line elements ( $\mathbf{v}^{n+1}$  in the above equation) as the FOWT undergoes both translation and rotation motions. A slightly different formulation is needed to handle frame transformation, that is, transforming a given quantity from one coordinate system to another. These transformations do not change the physical position of the point or body of interest, and thus are known as passive transformations. They are encoded into what is known as transformation matrix  $\underline{\mathbf{Q}}_{ab}$ , which transforms entities from frame 'a' to 'b':

$$\mathbf{v}_b = \underline{\mathbf{Q}}_{ab} \mathbf{v}_a, \quad \underline{\mathbf{T}}_b = \underline{\mathbf{Q}}_{ab} \underline{\mathbf{T}}_a \underline{\mathbf{Q}}_{ab}^T \quad (\text{B.4})$$

Rotation matrices are orthogonal with  $|\underline{\mathbf{Q}}| = 1$ , thus  $\underline{\mathbf{Q}}_{ba} = \underline{\mathbf{Q}}_{ab}^T$ . Transformation matrices are useful for switching between different coordinate systems as they hold a close relationship with rotation matrices: if 'b' is a reference frame derived after applying the rotation matrix  $\underline{\mathbf{R}}$  to the axes of frame 'a', then:

$$\underline{\mathbf{Q}}_{ab} = \underline{\mathbf{R}}^T \rightarrow \mathbf{v}_a = \underline{\mathbf{R}} \mathbf{v}_b, \quad \underline{\mathbf{T}}_a = \underline{\mathbf{R}} \underline{\mathbf{T}}_b \underline{\mathbf{R}}^T \quad (\text{B.5})$$

Thus, rotation matrices allow both physical rotations and frame transformations. So far, they have been kept in generic, abstract form: how they relate to actual geometric rotations is presented below.

### Euler angles

Introduced by Leonhard Euler to describe the orientation of a rigid-body with respect to a fixed frame, Euler angles are a set of three angular quantities chosen based on the fact that three chained elemental rotations always suffice to reach any target frame. Euler angles are indeed the angular amplitudes of each elemental rotation. Still, different conventions exist depending on the choice of the three rotation axes:

**Extrinsic rotations.** Rotations are performed about the axes of the original coordinate system, which is assumed to remain motionless. The resulting angles are known as "*Proper Euler*" angles.

**Intrinsic rotations.** Rotations are performed about the rotating axes, which change after each elemental rotation. For instance, a given axis  $\mathbf{e}$  is transformed onto  $\mathbf{e}'$  after the first elemental rotation and  $\mathbf{e}''$  after the second.

The resulting Euler angles depend not only on whether the rotation is extrinsic or intrinsic but also on the rotation chain sequence order. The most common compositions are z-y-x for extrinsic rotations and z-y'-x" for intrinsic rotations. The latter is known as "*Tait–Bryan*" composition and is the one used in the present project: yaw ( $\psi$ ) is the elemental rotation in z, pitch ( $\theta$ ) is in y' and roll ( $\phi$ ) in x". Because rotations are linear transformations, they can be applied in sequence:

$$\mathbf{v}' = \underline{\mathbf{R}}_\phi \underline{\mathbf{R}}_\theta \underline{\mathbf{R}}_\psi \mathbf{v} = \underline{\mathbf{R}}_{\{\phi, \theta, \psi\}} \mathbf{v} \quad (\text{B.6})$$

Given the three Tait–Bryan angles, the rotation matrix can be computed as:

$$\underline{\mathbf{R}}_{\{\phi, \theta, \psi\}} = \begin{bmatrix} c_\psi c_\theta & c_\psi s_\theta s_\phi - c_\phi s_\psi & s_\psi s_\phi + c_\psi c_\phi s_\theta \\ c_\theta s_\psi & c_\psi c_\phi + s_\psi s_\theta s_\phi & c_\phi s_\psi s_\theta - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \quad (\text{B.7})$$

Where  $s$  and  $c$  represent sine and cosine. Computing Euler angles from the rotation matrix is also possible:

$$\psi = \arctan\left(\frac{R_{21}}{R_{11}}\right), \quad \theta = \arctan\left(\frac{-R_{31}}{\sqrt{1 - R_{31}^2}}\right), \quad \phi = \arctan\left(\frac{R_{32}}{R_{33}}\right) \quad (\text{B.8})$$

The sign of the Euler angles is commonly defined according to the right-hand rule. In order for these angles to lead to unambiguous rotations, the range of  $\psi$  and  $\phi$  covers  $2\pi$  radians whereas  $\theta$  covers only  $\pi$ . Nonetheless, there are still some cases (when initial and final  $xy$  planes coincide) where Euler angles are not uniquely determined (gimbal lock). Other rotation representation approaches such as axis-angle and quaternions do not suffer from gimbal lock; the former is presented below.

### Axis-angle representation

Even though Euler angles are practical to represent the rotation of a floater, they are not as practical for describing the turbine rotor rotation. In such a case, the instantaneous axis of rotation is known, as is the rotated angle, which can be derived from the rotor angular speed. By Rodrigues' rotation formula, the rotation unit vector  $\mathbf{e}$  and angle  $\theta$  suffice to represent any rotation in 3D space:

$$\mathbf{v}' = (\cos \theta) \mathbf{v} + (\sin \theta) (\mathbf{e} \times \mathbf{v}) + (1 - \cos \theta) (\mathbf{e} \cdot \mathbf{v}) \mathbf{e} \quad (\text{B.9})$$

For some applications however, it is more suitable to transform the axis-angle representation into a rotation matrix:

$$\underline{\mathbf{R}}_{\{\mathbf{e}, \theta\}} = \begin{bmatrix} e_x^2 C + c_\theta & e_x e_y C - e_z s_\theta & e_x e_z C + e_y s_\theta \\ e_y e_x C + e_z s_\theta & e_y^2 C + c_\theta & e_y e_z C - e_x s_\theta \\ e_z e_x C - e_y s_\theta & e_z e_y C + e_x s_\theta & e_z^2 C + c_\theta \end{bmatrix} \quad (\text{B.10})$$

Where  $s$  and  $c$  represent sine and cosine, and  $C = 1 - c_\theta$ . Conversion from rotation matrix back to axis-angle representation is possible, although it requires special care with ambiguities.

## B.2. Implementation of new restraints

### B.2.1. Constant loads

As dull as it may sound, the *sixDoFRigidBodyMotion* library does not include any restraint that can represent the effect of a constant force and torque. Thus, a new restraint called `constantLoad` is created to fulfil this gap. This might turn out to be useful for defining loads from components that are not explicitly modeled, such as turbine-tower weight or aerodynamic thrust and torque. The latter can be practical for floater-only simulations where the overturning effect of the thrust force must be considered.

The user can use it by calling the `constantLoad` restraint in the *dynamicMeshDict* file:

```

1   restraints
2   {
3       exampleLoad // Restraint name (must be unique)
4       {
5           sixDoFRigidBodyMotionRestraint    constantLoad; // Restraint type
6           applicationPt    (0 0 0); // Force application point
7           movePt           true; // Application point moves with rigid body
8           force            (1 1 0); // Constant force (inertial frame)
9           torque           (0 0 1); // Constant torque (inertial frame)
10      }
11  }
```

Via the `movePt` entry, the user can choose whether the application point of the force moves along with the rigid body or is kept fixed in space. The moment introduced by the force is automatically taken into account. The user can also adjust the initial conditions through the `acceleration` and `torque` keywords to make them consistent with the applied loads; see section 5.1.2.

### B.2.2. Gyroscopic moment

Because of the extended range of motions of FOWTs compared to their fixed counterparts, gyroscopic loads can have a considerable effect on the platform motion. These loads arise from the non-linear term in the Euler equation of angular motion (equation 2.28). If the rigid-body model accounts for the rotor mass and inertia, then the gyroscopic moment is implicitly accounted for (e.g., in multibody or FEM simulations [6]).

However, in the present report, only the floater is modelled as a rigid-body, whereas the loads induced by the turbine (e.g., aerodynamic loads and weight) are explicitly included. Because of the motion in the extra DoFs, a correct assessment of the floating turbine behaviour requires careful consideration of gyroscopic loads, especially when the motion frequency closely matches that of the rotor [7]. These have already been identified as relevant even for single-DoF motions, where the combined rotor rotation and platform pitch motions yield a resulting gyroscopic moment that causes the FOWT to yaw [8].

#### Derivation

In order to account for the rotor inertia effects, a new restraint called `gyroscopicMoment` is created that can be used within the *sixDoFRigidBodyMotion* library. The present implementation is based on the derivations by Chen et al. [9] which model the FOWT by coupling two separate dynamic systems, namely the floater and rotor. They start off by defining the angular acceleration of the rotor relative to the inertial frame  $\dot{\omega}_R$ :

$$\omega_R = \Omega + \phi, \quad \dot{\omega}_R = \dot{\Omega} + \dot{\phi} + \phi \times \Omega \quad (\text{B.11})$$

Where  $\Omega$  is the rotor angular velocity relative to the floater. Recall that here  $\phi$  represents the floater angular DoFs presented in section B.1.1. The last term on the right-hand side of the above equation is referred to as Coriolis acceleration and is the main driver of the gyroscopic moment. After introducing the above expression into the governing equations of the coupled rigid body, an additional term emerges that corresponds to the sought gyroscopic moment from equation 2.28:

$$\mathbf{M}_{\text{gyro}} = - [\mathbf{J}_R \cdot \dot{\Omega} + \phi \times (\mathbf{J}_R \cdot \Omega)] \quad (\text{B.12})$$

Assuming the rotor rotation velocity remains unchanged (zero angular acceleration), then:

$$\dot{\Omega} = 0 \rightarrow \mathbf{M}_{\text{gyro}} = -\dot{\phi} \times (\mathbf{J}_R \cdot \Omega) \quad (\text{B.13})$$

### Implementation

So far, the frame within which the different variables are expressed has not been specified. Each one may be easier to express in a different frame, thus requiring transformation operations. The quantities in play are:

- Floater angular velocity, suitably defined in inertial frame  $\dot{\phi}_{(i)}$ .
- Rotor angular velocity, suitably defined in floater frame  $\Omega_{(f)}$ .
- Rotor inertia tensor, suitably defined in rotor frame  $\mathbf{J}_{R(r)}$ .

First, let's map the rotor inertia onto the floater frame. Because of the rotor's motion, its inertia in the floater frame will vary over time. Still, it remains unchanged in the rotor frame, thereby calling for the following transformation from equation B.4:

$$\mathbf{J}_{R(f)} = \mathbf{Q}_{rf} \mathbf{J}_{R(r)} \mathbf{Q}_{rf}^T \quad (\text{B.14})$$

Where  $\mathbf{Q}_{rf}$  is the transformation matrix from rotor to floater frame and coincides with the rotor rotation matrix  $\mathbf{R}_r$  that defines the rotor orientation from the floater frame. The latter matrix can be derived from its more comprehensive axis-angle representation by equation B.10. Assuming that the rotor axis is kept fixed with respect to the floater and that the rotor velocity  $\Omega$  is constant, then transformation B.14 is straightforward to apply. The gyroscopic moment in the floater frame then reads as:

$$\mathbf{M}_{\text{gyro}(f)} = -\dot{\phi}_{(f)} \times \left( \mathbf{R}_r \mathbf{J}_{R(r)} \mathbf{R}_r^T \cdot \Omega_{(f)} \right) \quad (\text{B.15})$$

Finally, the gyroscopic load is written in the inertial frame:

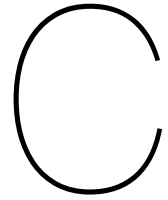
$$\mathbf{M}_{\text{gyro}(i)} = \mathbf{Q}_{fi} \mathbf{M}_{\text{gyro}(f)} = -\dot{\phi}_{(i)} \times \left[ \mathbf{Q}_{fi} \left( \mathbf{R}_r \mathbf{J}_{R(r)} \mathbf{R}_r^T \cdot \Omega_{(f)} \right) \right] \quad (\text{B.16})$$

This load is applied as an external restraint onto the rigid body equation. The user can use it by calling the `gyroscopicMoment` restraint in the `dynamicMeshDict` file:

```

1 restraints
2 {
3     turbineGyroscopic // Restraint name (must be unique)
4     {
5         sixDoFRigidBodyMotionRestraint    gyroscopicMoment; // Restraint type
6         rotationAxis    (1 0 0); // Rotor axis in floater frame
7         angularSpeed    1; // Rotor angular speed, in [rad/s]
8         inertiaMoment    (1 0 0 0 1 0 0 0 1); //Rotor inertia tensor in rotor frame
9     }
10 }
```

Although this restraint was constructed in the context of floating turbines, it can be applied to any arbitrary multi-body system in which one of the elements is not implicitly modeled.



# Scripts and templates

This chapter provides basic templates and scripts for some of the libraries used throughout the thesis. Although the complete libraries and simulation cases are available in [GitHub](#), novice users might benefit from these templates.

## C.1. waves2Foam

The relaxation zones and wave properties are defined within the *waveProperties.input* file, inside the *constant* folder. Below is a template for such a file, assuming Stokes' second-order wave at the inlet and undisturbed flow at the outlet.

```
1 FoamFile
2 {
3     version      2.0;
4     format       ascii;
5     class        dictionary;
6     object       environmentalProperties;
7 }
8 // * * * * * //
9 // Sea level
10 seaLevel      0.00;
11 seaLevelAsReference true;
12
13 // A list of the relaxation zones in the simulation.
14 // The parameters are given in <name>Coeffs below.
15 relaxationNames (inlet outlet);
16
17 // Initialize free-surface according to this condition
18 initializationName inlet;
19
20 inletCoeffs
21 {
22     // Wave theory to be applied at boundary "inlet"
23     // and in relaxation zone "inlet"
24     waveType     stokesSecond;
25
26     // Ramp time of 0s, default value is period
27     // Free-surface initialization requires Tsoft = 0
28     Tsoft        0.0;
29
30     // Water depth at the boundary and in the relaxation zone
31     depth        150.0;
32
33     // Wave period
34     period       8.0;
35
36     // Phase shift in the wave
37     phi          0.000000;
38 }
```

```

39 // Wave propagation direction
40 direction (1.0 0.0 0.0);
41
42 // Wave height
43 height 0.1;
44
45 // Print debug info
46 debug false;
47
48 // Specifications on the relaxation zone shape and relaxation scheme
49 relaxationZone
50 {
51 // Scheme: empty | spatial
52 relaxationScheme Spatial;
53 // Shape: Rectangular | Semi-cylindrical | cylindrical | frozen
54 relaxationShape Rectangular;
55 // Sloped beach
56 beachType Empty;
57 // Type: INLET | OUTLET
58 relaxType INLET;
59 //StartX and EndX define the corners of a box containing the relaxation zone
60 startX (0.0 -150.0 0.0);
61 endX (100.0 10.0 0.1);
62 // Relaxation direction
63 orientation (1.0 0.0 0.0);
64 // By default, an exponential relaxation weight is used.
65 // See Manual page 19 for other options
66 }
67 };
68
69 outletCoeffs
70 {
71 // Current which is uniform over the depth
72 waveType potentialCurrent;
73 // Current velocity
74 U (0 0 0);
75 // Ramp time
76 Tsoft 0.1;
77
78 relaxationZone
79 {
80 relaxationScheme Spatial;
81 relaxationShape Rectangular;
82 beachType Empty;
83 relaxType OUTLET;
84 startX (200.0 -150.0 0.0);
85 endX (400.0 10.0 0.1);
86 orientation (1.0 0.0 0.0);
87 }
88 };
89
90 // * * * * * //

```

## C.2. dynamicMeshDict

The definition of both the rigid-body and morphing mesh is done within the *dynamicMeshDict* file, inside the *constant* folder. This section presents a template for such a file (for 6-DoF rigid body and morphing mesh). All values there are chosen arbitrarily. If the user wishes to know which options are available (e.g., types of diffusivityModel), just replace the keyword with a dummy entry (e.g., bannana) and run the simulation. OpenFOAM won't start the solver, but output the available options for that specific keyword.

```

1 FoamFile
2 {
3   version      2.0;
4   format       ascii;
5   class        dictionary;
6   object       motionProperties;
7 }

```

```

8 // * * * * * * * * * * * * * * * * * * * * //
9
10 // Select mesh motion type
11 dynamicFvMesh dynamicMotionSolverFvMesh;
12
13 // 6-DoF rigid-body library and solver
14 motionSolverLibs ("libsixDoFRigidBodyMotion.so");
15 solver sixDoFRigidBodyMotion;
16
17 // Motion diffusivity
18 // diffusivity [distanceType] [diffusivityModel] [distance] ([patch])
19 diffusivity quadratic inverseDistance (bodyPatch);
20
21 sixDoFRigidBodyMotionCoeffs
22 {
23     // *** MESH PARAMETERS ***
24     // Name of the rigid-body surface patch
25     patches          (bodyPatch);
26     // Surface-normal distance, no mesh deformation below it
27     innerDistance    0.05;
28     // Surface-normal distance, no mesh deformation above it
29     outerDistance    0.5;
30
31     // *** RIGID BODY PROPERTIES ***
32     // Gravity vector (for weight calculation)
33     g                (0 0 -9.8065);
34     // Total mass
35     mass             100;
36     // Moment of inertia J (in principal axes)
37     momentOfInertia (0.5 0.5 1);
38
39     // *** INITIAL CONDITIONS ***
40     // Center of mass (initial position)
41     centreOfMass     (1 0 0);
42     // Initial linear velocity
43     velocity          (0 0 0);
44     // Initial linear acceleration
45     acceleration      (0 0 0);
46     // Center of rotation (default value is CoM)
47     centreOfRotation (1 0 0);
48     //Initial rotation tensor Q (ZYX convention)
49     orientation       (1 0 0 0 1 0 0 0 1);
50     // Initial angular momentum (w*J)
51     angularMomentum (0 0 0);
52     // Initial torque
53     torque            (0 0 0);
54
55     // *** RELAXATION AND DAMPING ***
56     // Relaxation parameter (phi): a_new = a_old + phi (a_new - a_old)
57     accelerationRelaxation 1.0;
58     // Damping parameter (f): a *= f
59     accelerationDamping 1.0;
60
61     // *** OUTPUT CONTROL ***
62     // Write motion to log
63     report            on;
64     // Write motion to time folders
65     reportToFile      on;
66
67     // *** SOLVER CONTROL ***
68     solver
69     {
70         // Solver options: Newmark | CrankNicolson | symplectic
71         type           Newmark;
72         // Velocity integration coefficient, default is 0.5
73         gamma          0.5;
74         // Position integration coefficient, default is 0.25
75         beta           0.25;
76     }
77
78     // *** RESTRAINTS ***

```

```

79 // See folder 'src/sixDoFRigidBodyMotion/sixDoFRigidBodyMotion/restraints'
80 restraints
81 {
82     // Restraints (i.e., external forces)
83 }
84
85 // *** CONSTRAINTS ***
86 // See folder 'src/sixDoFRigidBodyMotion/sixDoFRigidBodyMotion/constraints'
87 constraints
88 {
89     // Constraints (i.e., restricted DoFs)
90     // Options are: axis | line | orientation | plane | point
91 }
92
93 }
94
95 // * * * * * //

```

### C.3. Harmonic turbine motion

When implementing the turbine prescribed motions in Chapter 6, the position and velocities of the actuator elements from *turbinesFoam* were successfully verified against the following MATLAB script. The script assumes a rotor with constant angular speed, subject to harmonic motions in all six degrees of freedom.

```

1 %% Read actuator element file info
2 filename = "turbineData.csv";
3 raw_data = table2array(readtable(filename));
4 % Time vector
5 time = raw_data(:,1);
6 % AE position from turbinesFoam
7 pos_TF = [raw_data(:,3) raw_data(:,4) raw_data(:,5)];
8 % AE pelocity from turbinesFoam
9 vel_TF = [raw_data(:,7) raw_data(:,8) raw_data(:,9)];
10
11 %% Turbine data
12 axis_0 = [-1 0 0]; % Rotation axis
13 origin_0 = [0.2 0.2 0.2]; % Rotor hub positioib
14 rot_angle_0 = 0; %Azimuth offset
15 TSR = 6; % Tip-speed ratio
16 R = 0.45; %radius
17 U = 10; % Free-stream velocity
18 omega = U*TSR / R; % Angular speed
19
20 % Vector of azimuth angles
21 rot_angle = rot_angle_0*ones(1,length(time)) + omega*time;
22
23 %% Harmonic motion data
24 % Translations: [surge sway heave]
25 traFrequency = 2*pi*[2 2 2]; % Frequency [rad/s]
26 traAmplitude = [0.25 0.25 0.25]; % Amplitude [m]
27 % Rotations: [roll pitch yaw]
28 rotFrequency = 2*pi*[2 2 2]; % Frequency [rad/s]
29 rotAmplitude = (pi/180)*[45 45 45]; % Amplitude [rad]
30 % Initial AE position
31 currentPos = [0 0 0.02825];
32 % Initial orientation (Euler angles)
33 orientation_0 = (pi/180)*[15 15 15];
34 % Initial rotation center
35 rotCenter_0 = [-0.2 -0.2 -0.2];

```

```

36
37 %% Harmonic functions
38 refTrans = traAmplitude.*sin(traFrequency.*time);
39 refVel = traFrequency.*traAmplitude.*cos(traFrequency.*time);
40 refTheta = orientation_0 + rotAmplitude.*sin(rotFrequency.*time);
41 refOmega = rotFrequency.*rotAmplitude.*cos(rotFrequency.*time);
42
43 %% Initialise
44 axis = axis_0;
45 origin = origin_0;
46 rot_angle_prev = rot_angle_0;
47 rotCenter = rotCenter_0;
48 orientation = orientation_0;
49 % Rotation matrix from Euler angles
50 R = eul2rotm([orientation(3) orientation(2) orientation(1)])';
51 % Move point towards origin
52 currentPos = currentPos + origin;
53 % Rotate initial position according to initial rotation
54 currentPos = (currentPos-rotCenter)*R + rotCenter;
55
56 %% Advance in time
57 for i=1:length(time)
58     % localPos: current position wrt rotation center
59     % Un-rotate the point using R and rotation center from previous t
60     unrotPos = (currentPos-rotCenter)*R' + rotCenter;
61
62     % Perform turbine rotation
63     delta_angle = rot_angle(i)-rot_angle_prev;
64     turbinePos = axisRot(origin, axis, delta_angle, unrotPos);
65     rot_angle_prev = rot_angle(i);
66     origin = origin_0 + refTrans(i,:);
67
68     % Compute R from current timestep
69     orientation = refTheta(i,:);
70     R = eul2rotm([orientation(3) orientation(2) orientation(1)])';
71
72     % Position wrt rotationCenter
73     localPos = turbinePos - rotCenter;
74     localPos = localPos*R;
75
76     % Move the rotation center accordingly
77     rotCenter = rotCenter_0 + refTrans(i,:);
78
79     % current position = local position + rotCenter + translation
80     currentPos = rotCenter + localPos;
81     % Predicted AE position
82     pos(i,:) = currentPos;
83
84     % Compute linear and rotation velocities
85     % v = wr, both w and r given in moving frame
86     rotVel = cross(refOmega(i,:)*R,localPos);
87     % Predicted AE velocity
88     vel(i,:) = refVel(i,:) + rotVel;
89 end
90 % Now compare 'posTF' with 'pos', and 'velTF' with 'vel'

```

## C.4. IOdictionary

OpenFOAM's object registry can be thought of as a large database that holds (i.e. *registers*) references to various objects that can be accessed during run-time. Many types of entities are compatible with it, with the `IOdictionary` being of special interest since it can hold multiple attributes in it. In a nutshell, it can make entities that would be restricted to a certain class globally available, pretending they are global variables. In the thesis, this tool was used to communicate information between *sixDoFRigidBodyMotion* and *turbinesFoam*.

To exemplify the process, consider the rigid-body motion parameters. They are contained within *sixDoFRigidBodyMotion*, but must be accessed by *turbinesFoam* to move the turbine accordingly. First, the `IOdictionary` is created inside *sixDoFRigidBodyMotion.C* via the following function, which is called upon object initialization:

```

1 void Foam::sixDoFRigidBodyMotion::createDict()
2 {
3     // Create dictionary if it has not been created before
4     if(!time_.foundObject<IOdictionary>("sixDoFMotion"))
5     {
6         dictionary motionDict;
7         //mesh.thisDb().store
8         time_.store
9         (
10            new IOdictionary
11            (
12                IOobject
13                (
14                    "sixDoFMotion",
15                    time_.timeName(),
16                    time_,
17                    IOobject::NO_READ,
18                    IOobject::AUTO_WRITE
19                ),
20                motionDict
21            )
22        );
23        Info << "Rigid body IOdictionary 'sixDoFMotion' created" << endl;
24        updateDict();
25    }
26 }

```

The `IOdictionary` is updated every time the rigid-body solver is executed:

```

1 void Foam::sixDoFRigidBodyMotion::updateDict()
2 {
3     if(time_.foundObject<IOdictionary>("sixDoFMotion"))
4     {
5         // Open and write
6         const dictionary& motionDict =
7             time_.lookupObject<IOdictionary>("sixDoFMotion");
8
9         // Update for the motion solver
10        dictionary updateDbDictionary = &motionDict;
11        updateDbDictionary.set("centreOfRotation", centreOfRotation());
12        updateDbDictionary.set("orientation", orientation());
13        updateDbDictionary.set("velocity", v());
14        updateDbDictionary.set("omega", omega());
15
16        // Needed to update the IOdictionary
17        const_cast<dictionary& > (motionDict) = updateDbDictionary;
18
19        Info << "Updating 'sixDoFMotion' IOdictionary" << endl;
20    }
21 }
22 }

```

Finally, the contents of the `IOdictionary` are accessed by *turbinesFoam*:

```

1 void Foam::fv::actuatorLineSource::readRigidBodyDict(const fvMesh& mesh)

```

```
2 {
3 // If rigid body dictionary exists
4 if(mesh.time().foundObject<IOdictionary>("sixDoFMotion"))
5 {
6 // Access dictionary
7 const dictionary& motionDict = mesh.time().lookupObject<IOdictionary>("sixDoFMotion");
8 // Save its reference
9 rigidBodyDict_ = motionDict;
10 if (debug)
11 {
12 Info << "Chosen 'rigidBody' motion type for " << name_ << "." << endl;
13 };
14 }
15 else
16 {
17 rigidBodyMotionActive_ = false;
18 Info << "Rigid body IOdictionary ('sixDoFMotion') could not be accessed." << endl;
19 Info << "Motion based on rigid body is thus disabled." << endl;
20 }
21 }
```

# References

- [1] Tobias Holzmann. “Mathematics, numerics, derivations and OpenFOAM®”. In: *Loeben, Germany: Holzmann CFD* (2016).
- [2] Suraj S Deshpande, Lakshman Anumolu, and Mario F Trujillo. “Evaluating the performance of the two-phase flow solver interFoam”. In: *Computational science & discovery* 5.1 (2012), p. 014016.
- [3] Lionel Gamet et al. “Validation of volume-of-fluid OpenFOAM® isoAdvector solvers using single bubble benchmarks”. In: *Computers & Fluids* 213 (2020), p. 104722.
- [4] Johan Roenby, Henrik Bredmose, and Hrvoje Jasak. “A computational method for sharp interface advection”. In: *Royal Society open science* 3.11 (2016), p. 160405.
- [5] Bjarke Eltard Larsen, David R Fuhrman, and Johan Roenby. “Performance of interFoam on the simulation of progressive waves”. In: *Coastal Engineering Journal* 61.3 (2019), pp. 380–400.
- [6] Michael Borg, Maurizio Collu, and Athanasios Kolios. “Offshore floating vertical axis wind turbines, dynamics modelling state of the art. Part II: Mooring line and structural dynamics”. In: *Renewable and Sustainable Energy Reviews* 39 (2014), pp. 1226–1234.
- [7] Pierre Blusseau and Minoo H Patel. “Gyroscopic effects on a large vertical axis wind turbine mounted on a floating structure”. In: *Renewable Energy* 46 (2012), pp. 31–42.
- [8] Jason Jonkman and Walter Musial. *Offshore code comparison collaboration (OC3) for IEA Wind Task 23 offshore wind technology and deployment*. Tech. rep. National Renewable Energy Lab. (NREL), Golden, CO (United States), 2010.
- [9] Jia-hao Chen, Ai-guo Pei, Peng Chen, and Zhi-qiang Hu. “Study on Gyroscopic Effect of Floating Offshore Wind Turbines”. In: *China Ocean Engineering* 35.2 (2021), pp. 201–214.