Turbine controllers

Introduction

Turbines often rely upon control systems to regulate their speed and the power they generate. Control systems are supported in an OrcaFlex turbine model through external functions.

This document introduces the types of turbine control system that can be modelled and discusses several example external functions written in Python. A more general discussion concerning the modelling of wind turbines can be found on the website:

https://www.orcina.com/SoftwareProducts/OrcaFlex/Examples/

Note, these examples are written to work with OrcaFlex v11.0a and later.

External functions

External functions, and the API, are documented in detail in the OrcFxAPI reference documentation, which can be found on the website:

https://www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/OrcFxAPIHelp/

A general guide to using external functions, with a non-turbine specific example set, is also available through the website:
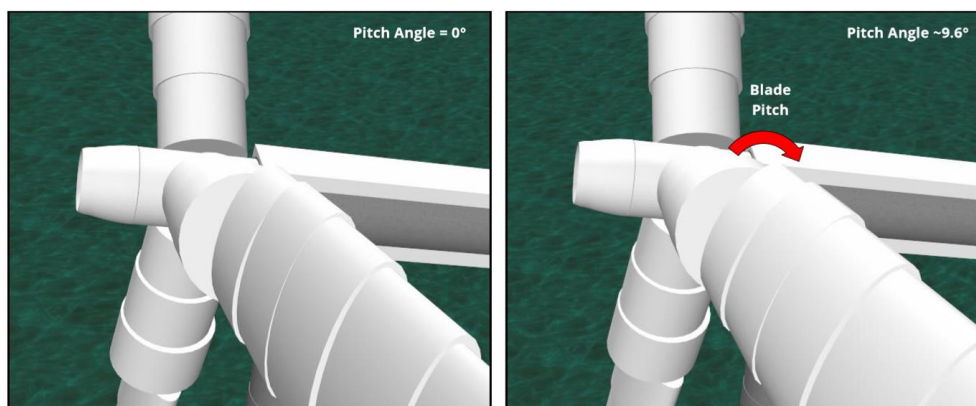
https://www.orcina.com/Support/

The below guide assumes your setup meets the minimum requirements to run a Python external function and you have a basic knowledge of working with them. If this is not the case, then you might like to review the above resources.

Types of turbine control

Pitch control

A pitch-regulated turbine varies the blade *pitch angle* to alter the aerodynamic torque on the rotor and thus control its speed and power. In an OrcaFlex turbine model, the pitch degree of freedom is the angle between the blade fitting frame, which is fixed relative to the hub, and its root frame.

Turbine controllers

Note the sign convention for blade pitch, for which a positive value defines an anticlockwise rotation about the blade z-axis, looking from root to tip, is contrary to the usual OrcaFlex convention.

OrcaFlex supports two **Pitch control modes**: `Common` and `Individual`. When operating in common mode, all blades share the same pitch value. In Individual mode, each blade has a unique pitch value.

To control the pitch degree of freedom, an external function is specified as the **Pitch controller** on the Blades page of the Turbine data form.

Common control mode

In common mode, each call to the specified external function should return the pitch value in *radians*, and its first two derivatives, that are to be used for all blades.

For a Python external function, this is done by assigning the pitch motion to the `info.StructValue` attribute in the external function's `Calculate` method, for example:

```
def Calculate(self, info):
    if info.NewTimeStep:
        info.StructValue.Value = pitch
        info.StructValue.Velocity = pitchDot
        info.StructValue.Acceleration = pitchDotDot
```

Individual control mode

In individual mode, each call to the specified external function should return the pitch value in *radians,* and its first two derivatives, for each blade separately.

For a Python external function, this is done by assigning each blade's pitch motion to an element of the list contained in the `info.StructValue` attribute, in the external function's `Calculate` method. The first blade's motions should be assigned to the first element (i.e. index 0) and so forth. For example, if the desired blade pitch values were contained in a list called `pitch`, and their derivatives in `pitchDot` and `pitchDotDot`, the external function might look like this:
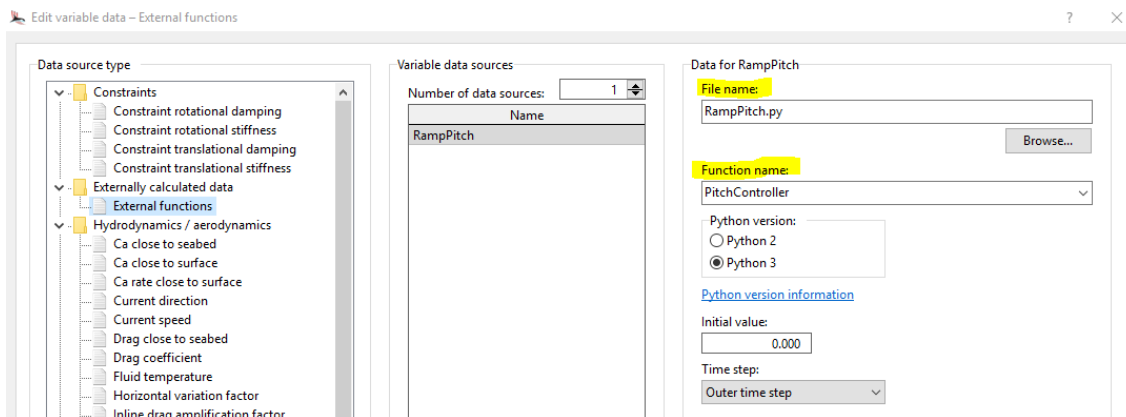
```
def Calculate(self, info):
    if info.NewTimeStep:
        for i in range(info.ModelObject.BladeCount):
            info.StructValue[i].Value = pitch[i]
            info.StructValue[i].Velocity = pitchDot[i]
            info.StructValue[i].Acceleration = pitchDotDot[i]
```
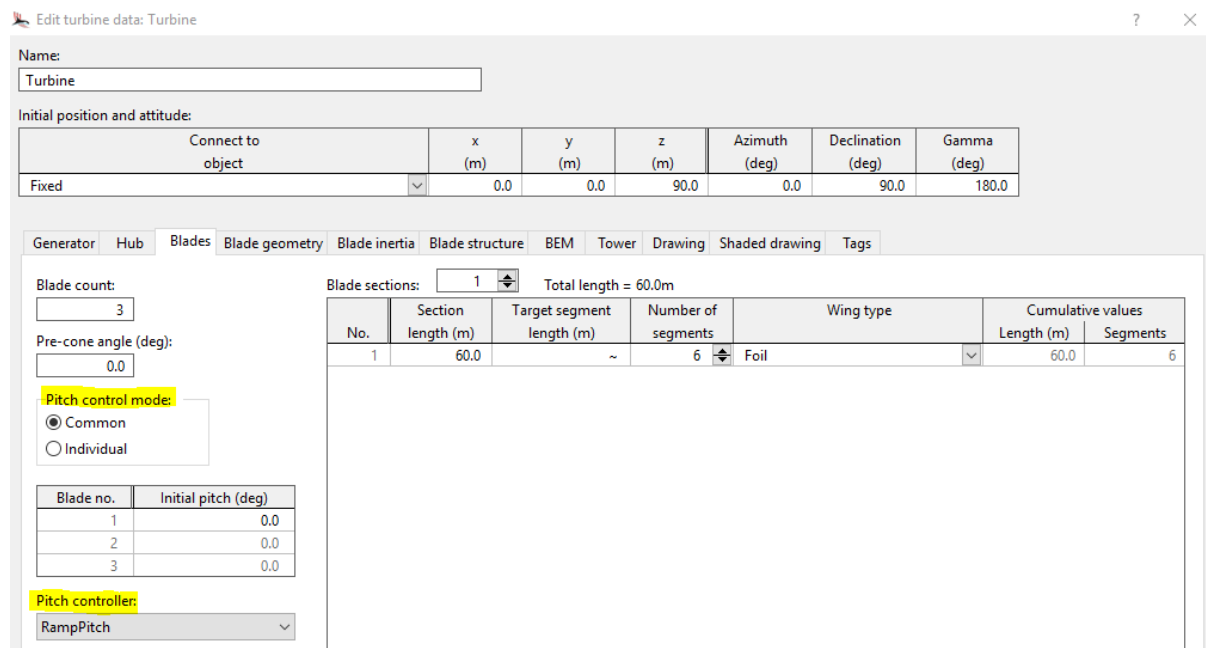
Turbine controllers

A trivial common control mode example: Ramp pitch

In this example, a very simple turbine is subjected to constant wind and no generator torque is applied (i.e. the rotor is initially allowed to freely accelerate). A Python external function is used to smoothly ramp the blade pitch, for all blades, up to 90 degrees. This reduces the aerodynamic torque on the rotor and eventually slows it almost to a stop.

Open `RampPitch.dat` in OrcaFlex, on the Variable data form there is a single external function named `RampPitch`. The data for this external function specifies: `RampPitch.py` as the **File name**, this is the name of the Python module containing the external function class definition; and `PitchController` as the **Function name**, this is the external function Python class name.

On the Blades page of the Turbine data form, the external function name is specified as the **Pitch controller** and `Common` is selected as the **Pitch control mode**.

Turbine controllers

Open `RampPitch.py` in a text editor, it contains the Python class which defines our external function. Because this is a very simple example, it only includes the `Calculate` method. This method is implemented to: check if it is being called at the beginning of a time step using `info.NewTimeStep`; calculate the pitch value, and its first two derivatives, assuming the form of a *logistic curve*; and return these values in the `info.StructValue` attribute. The logistic curve has the form of an "S" shape, which can be used as a simple ramping function, with equation:

$$f(t) = \frac{f_{max}}{1 + e^{-(t-t_0)}}$$

Back in OrcaFlex, run the simulation and open the associated default workspace (*Workspace>Use file default*). As expected, the turbine's main shaft angular velocity initially increases until the controller pitches the blade sufficiently to slow the rotor down.

A trivial individual control mode example: Individual pitch

In this example, the turbine does not rotate and a Python external function is used to impose harmonic blade pitch motion, with each blade being given unique amplitude.

Open `IndividualControl.dat` in OrcaFlex, on the Variable data form there is a single external function named `IndividualControl`. The data for this external function specifies: `IndividualControl.py` as the **File name** and `PitchController` as the **Function name**. On the Blades page of the Turbine data form, the external function name is specified as the **Pitch controller** and `Individual` is selected as the **Pitch control mode**.

Open `IndividualControl.py` in a text editor, it contains the Python class which defines our external function. Again, it only includes the `Calculate` method. This method is implemented to: check if it is being called at the beginning of a time step using `info.NewTimeStep`; calculate pitch values for each blade, and the first two derivatives, assuming harmonic form with an amplitude scaling factor unique to each blade; and return these values in the elements `info.StructValue` attribute.

Back in OrcaFlex, run the simulation and open the associated default workspace (*Workspace>Use file default*). Each blade pitches harmonically, with amplitude determined by the blade's unique scaling factor.

Turbine controllers

Generator torque control

Turbines also control the rotor speed by varying the torque applied by the generator. To control the applied torque, an external function can be specified as the **Generator controller** on the Generator page of the Turbine data form.

Unlike the pitch controller, the generator controller can operate in two modes: `specified torque` and `specified rotation`. To control the applied torque, the `specified torque` mode should be selected.
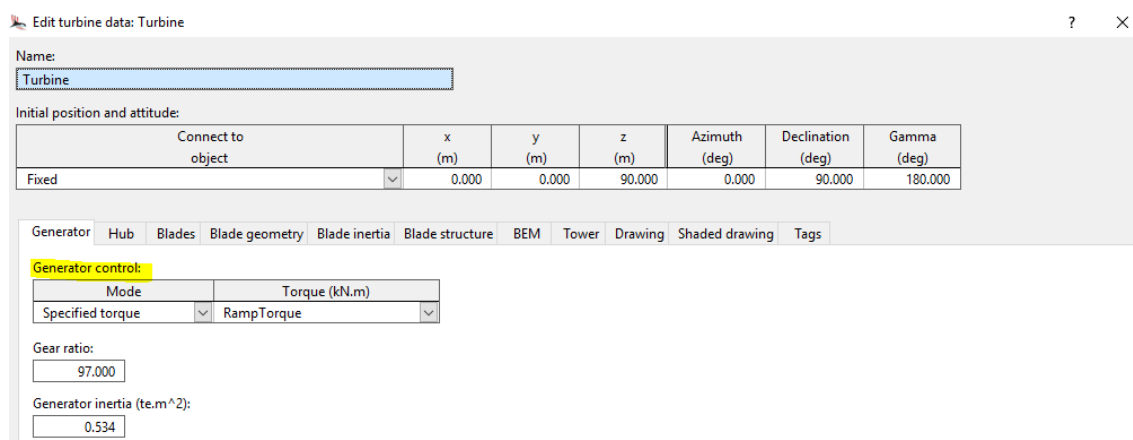
Each call to the specified external function should return the generator torque value. In a Python external function, this is done by assigning the torque to the `info.Value` attribute in the external function's `Calculate` method, for example:

```
def Calculate(self, info):
    if info.NewTimeStep:
        info.Value = torque
```

A trivial example: Ramp torque

In this example, a very simple turbine is subjected to no wind. A Python external function is used to smoothly ramp the generator torque up to 100 kN.m, accelerating the rotor as it does.

Open `RampTorque.dat` in OrcaFlex. On the Variable data form there is a single external function named `RampTorque`. The data for the external function specifies `RampTorque.py` as the **File name** and `TorqueController` as the **Function name**. On the Generator page of the Turbine data form, `Specified torque` is selected as the **Generator mode** and the external function name is specified as the **Generator torque controller**.

Open `RampTorque.py` in a text editor, it contains the Python class which defines our external function. Again, it only includes the `Calculate` method. This method is
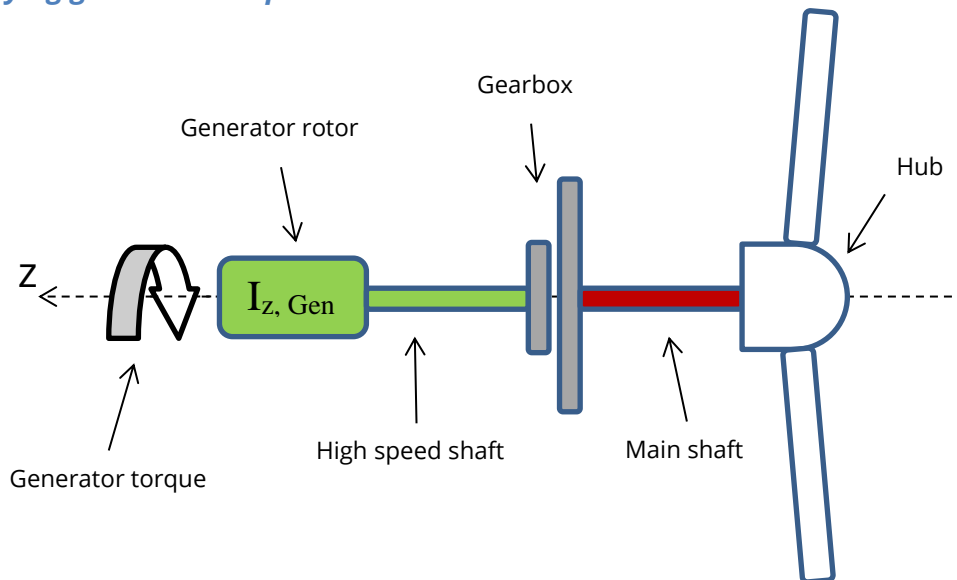
implemented to: check if it is being called at the beginning of a time step; calculate the torque value assuming the form of a *logistic curve*; and return this value in the `info.Value` attribute.

Note in this example, we have used the `OrcaFlexObject.UnitsConversionFactor` method to ensure the torque is returned in the unit system consistent with the model. Multiplying by this conversion factor converts a value from SI units into model units. The possible values for the base unit passed into it are detailed in the OrcFxAPI documentation.

Back in OrcaFlex, run the simulation and open the associated default workspace. As expected the generator torque is smoothly ramped up to 100 kN.m and the rotor accelerates.

Applying generator torque

The controller's torque is applied to the generator rotor using the standard OrcaFlex convention, i.e. clockwise when looking along the axis of rotation (or when viewing the turbine face-on). As is depicted above, the controller's torque is applied before the generator's rotational inertia, i.e. if the system is accelerating, and the generator's rotational inertia is non-zero, then the high speed shaft torque will not equal the applied torque.

If the gear ratio is positive, the main shaft will spin in the same direction as the generator. In this example the gear ratio is positive. So to make the rotor spin in the conventional clockwise turbine direction, the controller is setup to apply a *positive* torque, i.e. to put energy into the system. In a more realistic turbine model, if the gear ratio was also positive, the generator torque controller would apply a negative torque and take energy out of the system.

Turbine controllers

Generator motion control

The generator can also operate in the `Specified rotation` mode. This control mode is primarily used for testing and model verification. If you would like to use it, on the Generator page of the Turbine data form, `Specified motion` should be selected as the **Generator mode** and the external function name specified as the **Generator motion controller**. At each call of the specified external function, it should return the generator angle and its first two derivatives. This is done following the same pattern used for a common control mode pitch controller:

```
def Calculate(self, info):
    if info.NewTimeStep:
        info.StructValue.Value = angle
        info.StructValue.Velocity = angleVel
        info.StructValue.Acceleration = angleAccel
```

Python controller

This is a more realistic example of a conventional variable-speed, variable blade-pitch-to-feather controller, supported purely with a Python module. Contained within `PythonController.py` is a Python implementation of the NREL OC3 baseline controller (J. Jonkman, 2009), with the flexibility to optionally apply the OC3 Phase IV modifications (Jonkman, 2010).

The `5MWBaseline.dat` OrcaFlex data file is an example of the NREL OC3 5-MW baseline wind turbine in the fixed bottom configuration (J. Jonkman, 2009). This model uses the `PythonController.py` Python module to provide the generator torque and pitch control.

OrcaFlex model

Open `5MWBaseline.dat` in OrcaFlex. The model has an external function, `Controller` It specifies `PythonController.py` as the Python module containing the class that defines it. From this module, the `PythonController` Python class is selected. The `NREL-5MW` turbine object data sets this external function to provide the generator torque and pitch control. Setting the external function and turbine data is discussed in the previous sections.

To include this controller in a turbine model, the above is the minimum data that would need to be set. However, unless user specified object tags are optionally included, which are discussed below, the controller will, by default, not apply OC3 Phase IV modifications. Nor will the actuator model be used. If you are modelling a floating system, the OC3 Phase IV modifications are appropriate and the `FloatingSystem` object tag should be given.

Run the simulation and open the associated default workspace. The turbine is subjected to an incident wind that increases from zero to a constant value of 13 m/s over the build-up period. This wind speed is above the rated wind speed for this turbine and the pitch control system is thus required to maintain the rated rotor speed of 12.1 rpm (1.267 rad/s). Initially the rotor accelerates, exceeding the rated rotor speed, and the controllers then operate to correct it back to the rated value.

Note: the achieved generator power must be scaled by the turbine efficiency (94.4%) to recover the rated 5MW.

Turbine controllers

Modifying the behaviour with object tags

Objects in an OrcaFlex model maintain a set of name/value pairs known as tags. The external functions in the `PythonController.py` module are written to accept the following model tags from the object that calls them:

- `FloatingSystem`: if `True` the OC3 Phase IV modifications are made
- `UseActuator`: if `True` the blade pitch actuator model is enabled
- `ActuatorOmega`: the actuator's natural angular frequency (rad/sec)
- `ActuatorGamma`: the actuator's damping ratio

If the object tags are not specified, those that are interpreted as a Boolean are treated as being `False`. The actuator model is discussed in the below Actuator section.

This data is set on the Tags page of the Turbine data form, in this example it is set as:

Generator	Hub	Blades	Blade geometry	Blade inertia	Blade structure	BEM	Tower	Drawing	Shaded drawing	Tags

Name	Value
FloatingSystem	False
UseActuator	False
ActuatorOmega	188.0
ActuatorGamma	0.02

Here neither the OC3 Phase IV modifications nor the actuator model are enabled.

Python module

The `PythonController.py` module contains the class that defines both the pitch and generator torque control. If you open it in a text editor, the `PytonController` class can be found at the bottom of the module. The `Calculate` method of the this class is responsible for calling the `update` method of the `ControllerEngine` class. Depending on the controller type, it returns the associated control values through the correct `info` attributes.

The `ControllerEngine` class contains the implementation of the NREL controllers. Its `update` method is responsible for updating state and calculating all the controller values once per time step. The NREL OC3 baseline torque and pitch controllers share state and so their control values are calculated in a *single* instance of the `ControllerEngine` class, which is stored in their `controller` attributes.

The single `ControllerEngine` instance is shared between the `PitchController` and `TorqueController` instances using the `info.Workspace` attribute. This is a Python dictionary that is shared model wide. They access it using a key which is unique to the calling turbine object. This is done in the `Initialise` method of the parent class:

```
def Initialise(self, info):
    key = info.ModelObject.handle.value
    controller = info.Workspace.get(key, None)
```

Turbine controllers

```
if controller is None:
    controller = ControllerEngine(info)
    info.Workspace[key] = controller
self.controller = controller
```

Capabilities and Limitations

The `BaseController` parent class includes a `StoreState` method. This calls the `storeState` method of the `ControllerEngine`, which preserves the working state in the `info.StateData` attribute. It is called whenever a simulation is saved. Because the state is captured, and restored the next time the class is instantiated, part run simulation files can safely be continued and completed simulation files can safely be extended.

The external function uses the `OrcaFlexObject.UnitsConversionFactor` method to ensure the control values are returned consistently with the model unit system.

The NREL OC3 baseline controller expects the initial blade pitch to be zero. As such, the external function has the same requirement.

The external function is not written to work with the implicit variable time step integration scheme.

It is safe for multi-threading and models using it can be run in batch.

Turbine controllers

Bladed controller wrapper

Often a control's source code will not be made available and instead is compiled as a dynamic link library (DLL). In this example, we use an external function as a wrapper for the NREL OC3 baseline controller, bladed style, DLLs compiled by NREL. Two versions of the example are provided, covering both Python and native external functions.

Obtaining the DLLs

The DLLs, distributed by NREL, are available from:

https://github.com/OpenFAST/openfast/releases

1. Scroll down to `OpenFAST v2.0.0` and expand `Assets`
2. Download and open `windows_openfast_v2.0.0.zip`
3. Navigate to the `openfast_v2.0.0_binaries\DISCON_DLLS` folder
4. Depending on the bitness of your OrcaFlex, open either the `64bit` or `Win32` folder and copy the DLLs from within
5. Paste these into either the `Python controller` or `Native controller wrapper` folders of this `Bladed controller` example

OrcaFlex model

The 5MWBaselineWrapper.dat model is very similar to the previous example. Now specifying either the Python module, `BladedControllerWrapper.py`, or the native DLL, `BladedControllerWrapper.dll`, as the external files containing the `BladedController` external function.

Unlike the previous example, one object tag, `ControllerDLL`, must be specified, because this is used to tell the external function the filename of the target controller DLL.

Because the DLL is a complied version of the NREL OC3 baseline controller, when the simulation is run, the behaviour is almost identical to that of the previous example.

Modifying the behaviour with object tags

The external functions in the `BladedControllerWrapper.py` module are written to accept the following model tags from the object that calls them:

- `ControllerDLL`: filename of target DLL, given as a relative path
- `DLLCanBeShared`: if `True` then DLL is assumed safe for multi-threading
- `UseActuator`: if `True` the blade pitch actuator model is enabled
- `ActuatorOmega`: the actuator's natural angular frequency (rad/sec)
- `ActuatorGamma`: the actuator's actuator damping ratio
- `InputFile`: filename of input file passed to DLL, given as a relative path
- `AccelRefPosRrtTurbine`: optional, specifies the reference position at which the turbine's acceleration, passed in the swap array, is calculated. It should be a JSON string, from which `json.loads` will return a Python array-like object, e.g. `[-3.0,`

`0.0, 5.0]`. It represents a position vector, expressed relative and with respect to the turbine frame. If not given, the turbine origin is used.

If the object tags are not specified, those that are interpreted as a Boolean are treated as being `False`.

Unlike the pure Python controller, the OC3 Phase IV modifications cannot be enabled by specifying an object tag. Instead, you must tell the external function to target a DLL which has been compiled from a source in which the modifications have already been made. To change this example to use the modifications, the `ControllerDLL` tag can be set to `DISCON_OC3Hywind.dll`, depending on the bitness of your OrcaFlex.

In this example, `InputFile` is not used.

Python module

The Python controller example includes the `BladedControllerWrapper.py` module, which contains the `BladedController` class that defines both the pitch and generator torque control. The pitch and generator torque controllers will each have separate instances of this class. The behaviour switches on `info.DataName`. Their `controller` attribute contain an instance of the `Controller` class.

The `Controller` class contains the implementation of the wrapper. Its `update` method is responsible for calling the DLL, using the `callDLL` method, once per time step. When called, the DLL updates its state and calculates all the controller values.

The `avrSwap` swap array is used to communicate between the Python module and the DLL. Before the DLL is called, the appropriate elements in the swap array are populated with the values that are to be input to the calculation. This is then passed into the DLL when it is called. After it has been called, the DLL will have calculated the control values which are then read out of the swap array and returned through the external function.

Swap array

In this example, the `avrSwap` swap array is used to communicate with the DLL. The NREL DLL is written in the Bladed-style as documented in Appendix A of the Bladed User's Guide Version 3.51. The following elements are set by the wrapper before the DLL is called:

- `Record  1`: flag set to: 0 at first call; 1 for subsequent calls; -1 on final call
- `Record  2`: time since simulation started (s)
- `Record  3`: model time step (2)
- `Record  4`: current blade 1 pitch angle (rad)
- `Record 15`: generator power (W)
- `Record 20`: current generator angular velocity (rad/s)
- `Record 21`: current main shaft angular velocity (rad/s)
- `Record 23`: generator torque (Nm)[*]
- `Record 24`: yaw error (rad)[**]

- `Record 27`: current hub wind speed (m/s)[*]
- `Record 28`: flag set to: 0 common pitch mode; 1 individual pitch mode
- `Record 30`: current blade 1 root connection Ey moment (Nm)[*]
- `Record 31`: current blade 2 root connection Ey moment (Nm)[*]
- `Record 32`: current blade 3 root connection Ey moment (Nm)[*]
- `Record 33`: current pitch angle, blade 2 (blade 1 if common pitch mode) (rad)
- `Record 34`: current pitch angle, blade 3 (blade if common pitch mode) (rad)
- `Record 37`: nacelle yaw angle from North (rad)[**]
- `Record 49`: max number of characters of avcMsg, set to 1024
- `Record 50`: number of characters of accInfile, set to 1024 if no `InputFile` tag
- `Record 51`: max number of characters of avcOutname
- `Record 53`: turbine z acceleration (m/s^2)[*†]
- `Record 60`: current rotor angle (rad)
- `Record 61`: turbine blade count
- `Record 83`: turbine ry angular acceleration (rad/s^2)[*]

The following elements are set by the DLL and read by the wrapper:

- `Record 42`: blade 1 pitch demand (individual pitch mode) (rad)
- `Record 43`: blade 2 pitch demand (individual pitch mode) (rad)
- `Record 44`: blade 3 pitch demand (individual pitch mode) (rad)
- `Record 45`: blade pitch demand (common pitch mode) (rad)
- `Record 47`: generator torque (Nm)[*]

[*]When read/written converted to/from the OrcaFlex sign convention and the model's unit system.

[†]When the AccelRefPosRrtTurbine tag is given, the acceleration at the specified reference position is passed in this record.

[**]Range jumped suppressed.

Note: different control DLLs, written in same Bladed-style as this example, might need different elements to be populated or read by the wrapper. DLLs written to conform to later versions of Bladed use an extended swap array, with a different specification, and the wrapper code will need to be modified to be compatible with them.

Native wrapper

The native controller example contains the `BladedControllerWrapper.dll`, a compiled, 64 bit, native external function DLL. This is can be used in exactly the same way as the Python module and the above documentation is just as relevant in this case.

The advantage of the native wrapper is that it is not subject to the Python Global Interpreter Lock (GIL). The GIL can significantly impede performance with when using multi-threading to run models in parallel, e.g. when using OrcaFlex batch processing.

Turbine controllers

The disadvantage is the source code is harder to read, edit, and needs to be complied before it can be used. The source code is distributed with the example in the `scr` folder. To help compile the DLL, the `build` folder contains both a Microsoft Visual Studio project and a MSYS2 makefile.

Capabilities and Limitations

Unlike the pure Python controller, the wrapper example does not capture state when the model is saved. This means if you continue, extend or restart a simulation file, it will do so with fresh controller state.

The external function uses the `OrcaFlexObject.UnitsConversionFactor` method to ensure the control values are returned consistently with the model unit system.

The NREL OC3 baseline controller expects the initial blade pitch to be zero. As such, the external function has the same requirements.

The external function is not written to work with the implicit variable time step integration scheme.

The wrapper is written to support the OC3 DLLs compiled by NREL. Other DLLs, written to a consistent specification, might need extra elements of the swap array to be populated or read from.

The wrapper is intended to support a Bladed-style DLL as documented in Appendix A of the Bladed User's Guide Version 3.51. It will need to be modified to support later specifications.

It is safe for multi-threading and models using it can be run in batch. However, the `DLLCanBeShared` object tag must be set correctly. If `DLLCanBeShared` is `False` then for each model in the batch, a distinct copy of the DLL, with a unique name, is created which means that each of those DLLs have their own private global variables.

The bitness of the target DLL must match that of your OrcaFlex.

Using the Bladed controller wrapper with the NREL ROSCO control DLL

The external function in this example can be used as a wrapper for the NREL ROSCO control DLL.

https://github.com/nrel/rosco

To use it, the `InputFile` object tag should now be specified. The `InputFile` tag should give the path, relative to the model directory, of the DLL's input file.

Please note, the wrapper populates the swap array sufficiently to support the control modes used in the examples available on the Orcina website. If other controller modes are required, the wrapper might need to be further extended. The control modes, which are enabled in the controller input file, known **not to be supported** include: `Flp_Mode` and `OL_Mode`.

Turbine controllers

Debugging the NREL ROSCO control DLL

Unfortunately, if there is an error in the ROSCO DLL it can sometimes lead to OrcaFlex terminating without displaying an error message. For example, this can happen when the ROSCO DLL is unable to locate a required dependency (e.g. the input file or perf. file) or there is bitness mismatch between the ROSCO DLL and OrcaFlex. If this happens, a simple way to debug the issue it to run the OrcaFlex model through the API using a very simple Python script containing the below code snippet:

```
import OrcFxAPI
model = OrcFxAPI.Model('MyFile.dat')
model.RunSimulation()
```

If this script is run from the command line, then you should recover an error message detailing the issue and allowing you to resolve it.

Python actuator class

The `PythonController.py` and the Python/native wrappers contain an `Actuator` class. This class can optionally be used to include blade pitch actuator dynamics.

It is enabled by including the `UseActuator` object tag, with the value set to `True`, on the tags page of the turbine data form. If it is enabled, the `ActuatorOmega` and `ActuatorGamma` object tags, which specify the actuator natural frequency and damping ratio, must also be included.

The model approximates the actuator dynamics as a second order system:

$$\ddot{x} + 2\omega\gamma\dot{x} + \omega^2 x = \omega^2 u$$

Where u is the input blade pitch demand, x is the output blade pitch angle, ω is the actuator natural frequency and x is the actuator damping ratio. At each time step, this system is analytically solved assuming the input blade pitch demand changes linearly over the time step.

The actuator model has the advantage of turning a blade pitch demand (i.e. angle only) into the blade pitch motion (i.e. include the first two derivatives).

Bibliography

J. Jonkman, S. B. (2009). *Definition of a 5-MW Reference Wind Turbine for Offshore System Development* .

Jonkman, J. (2010). *Definition of the Floating System for Phase IV of OC3.*