



ROSCO Toolchain Tutorial

Nikhar J. Abbas

Alan Wright, Lucy Pao, Dan Zalkind

August 10, 2020



1 Preparatory Material

2 Theory Overview

3 Implementation Overview

4 Using the ROSCO Toolbox

5 Some rules of thumb

6 Tuning Example/Walk-through

7 Future Work, Acknowledgements, Etc..

Preparatory Material

The preparatory slides are meant to help prepare for the tutorial. The outlined steps should be completed before the tutorial if you wish to follow along on your own computer, otherwise, you can skip to slide 8.

Preparatory Material - Intro

Four Necessary Codes:

1. OpenFAST (Wind Turbine Simulation): <https://github.com/openfast/openfast>
2. WISDEM (OpenFAST file parsing, BEM solvers): <https://github.com/wisdem/wisdem>
3. ROSCO controller (controller): <https://github.com/NREL/ROSCO>
4. ROSCO toolbox (controller tuning): https://github.com/NREL/ROSCO_toolbox

Installation and Compiling:

- OpenFAST needs to be compiled individually.
- WISDEM and the ROSCO toolbox are both python-based codes. These can be installed using Anaconda or by cloning their respective repositories.
- ROSCO is Fortran-based and a compiled binary is needed. This is generally made using CMake or Anaconda

Fortunately, we can install them all using a few simple commands!

- Please note that these are all very dynamic codes. To compile these for long term use, please refer to their respective repositories.

Anaconda:

The Anaconda platform is used to manage python packages and can be downloaded at: <https://www.anaconda.com/>. If you do not have it, download and install this before proceeding. PLEASE install the correct anaconda for your operating system.

Preparatory Material – I

Setup Anaconda

1. Open a new terminal window (mac/linux) or command/anaconda prompt (windows). An “Anaconda Powershell Prompt” will cause errors in step 6, but a work-around is provided.
2. Navigate to your preferred folder for this tutorial, we will call it `<terminal_dir>`.

3. Setup an anaconda environment

```
conda config --add channels conda-forge
conda create -y --name rosco-env python=3.8 git
conda activate rosco-env
```

Download the tutorial repository

4. A git repository has been created to ease the workflow of this tutorial, lets start by cloning that and navigating to it (this will take a few minutes).

```
git clone https://github.com/nikhar-abbas/ROSCO_tutorial.git
cd ROSCO_tutorial
```

Preparatory Material – II

Install the ROSCO toolbox with WISDEM and the ROSCO controller

5. We will install [WISDEM](#) and [ROSCO](#) using anaconda, and the [ROSCO toolbox](#) using pip. It will take a few minutes for everything to download.

Note: The select few Windows 32-bit users will need to follow the non-conda-forge based WISDEM and ROSCO installation instructions in their respective git repositories.

```
conda install -y wisdom
conda install -y rosco
pip install rosco-toolbox
```

Move the ROSCO controller binary

6. The controller binary was installed in step 5. We do not *need* to move it, but will in this tutorial for simplicity.

Note: Again, you DO NOT need to do this. Only one binary is needed for all of your controller needs, and must only be pointed to by `DLL_FileName` in the ServoDyn input file.

On a mac/linux

```
cp $CONDA_PREFIX/lib/libdiscon.* BAR_00/libdiscon.b
```

On Windows 64-bit*

```
copy %CONDA_PREFIX%\Library\Lib\libdiscon.dll BAR_00\ServoData\libdiscon.b
```

On Windows 32-bit*

```
copy <ROSCO_path>\build\libdiscon.dll BAR_00\ServoData\libdiscon.b
```

*NOTE-for-Windows: If `%CONDA_PREFIX%` gives an error, replace it with the environment location provided by `conda info`

Preparatory Material – III

Download OpenFAST

7. OpenFAST is needed to run simulations. If you are on Windows you DO NOT need to do this step – windows executables were distributed with the tutorial's git repo.

For Mac/Linux only:

```
conda install -y openfast
```

Testing, Testing: 1, 2, 3.

12. From your terminal, within the roscot-env virtual environment, enter

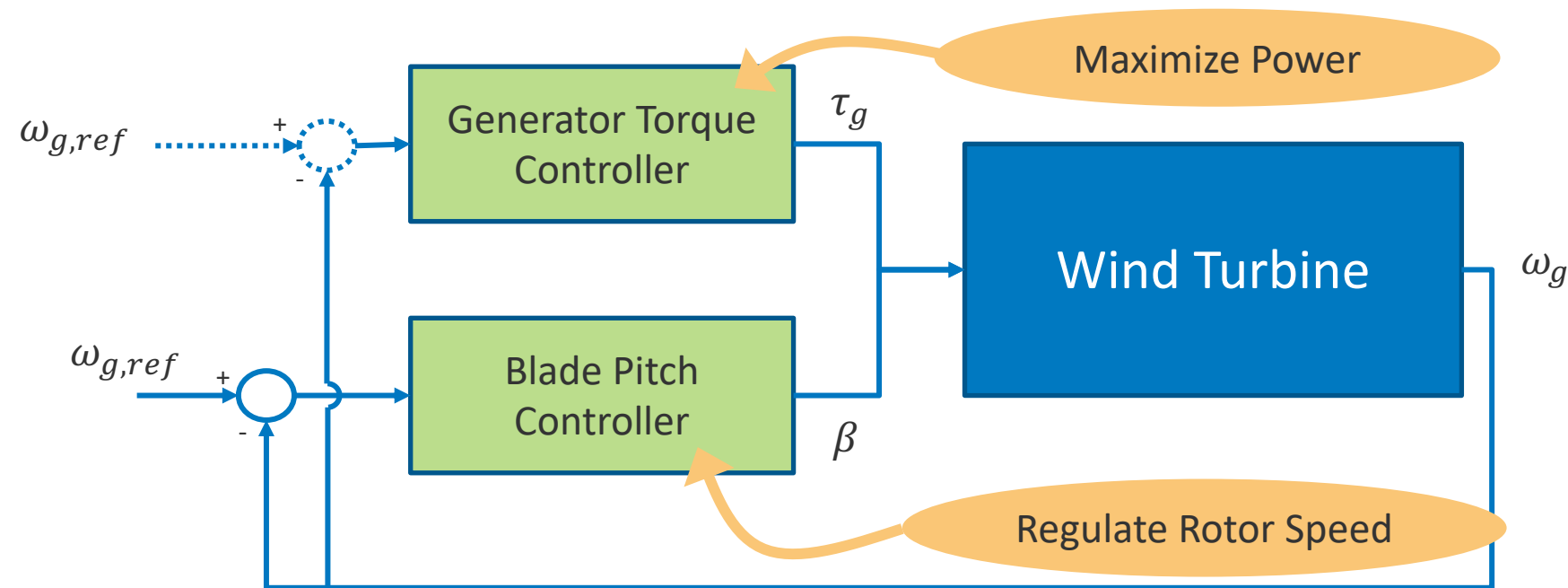
```
python run_ROSCOtuning.py  
python plot_ROSCOresults.py
```

Congratulations! If you have made it this far, you have gotten all of the tools necessary for execution of the ROSCO toolbox up and running and are ready for the tutorial.

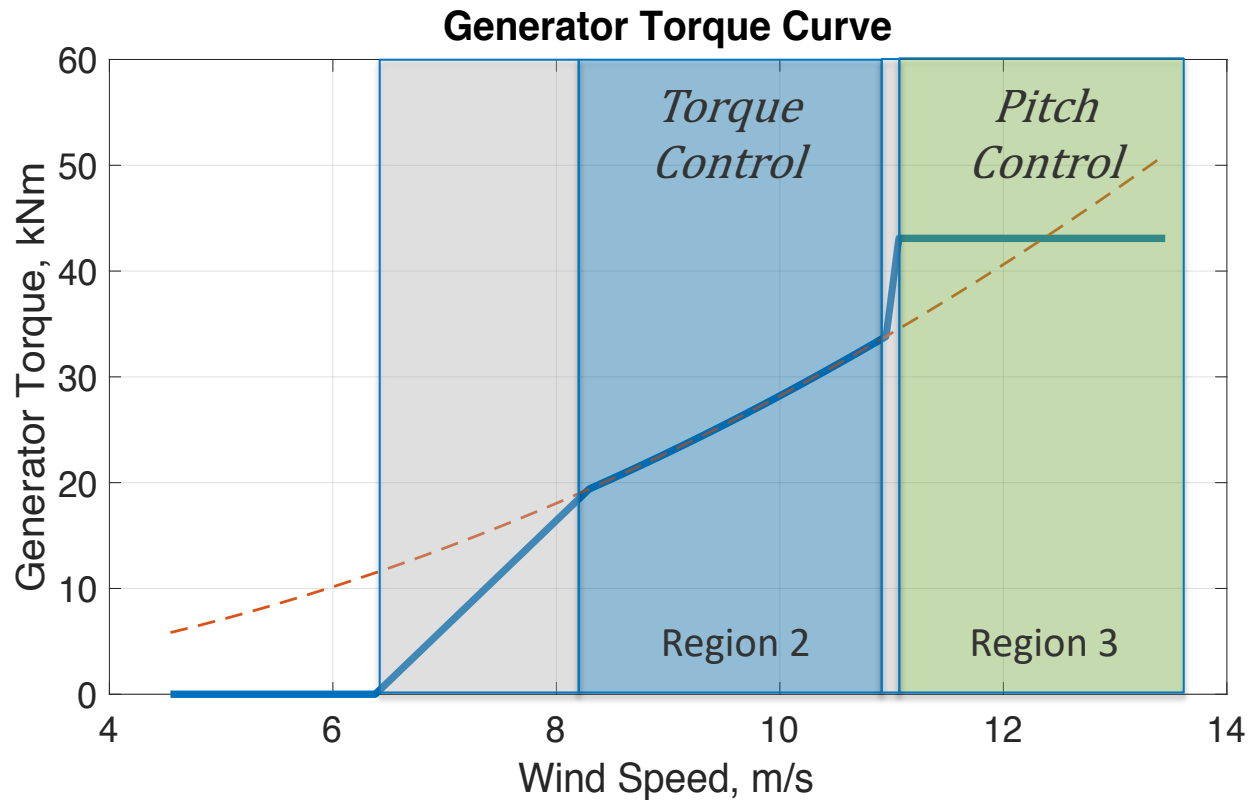
Theory Overview

Some very high-level theory of a wind turbine controller and how it relates to ROSCO

Theory – The standard controller



Theory – The standard controller



Theory – The standard controller contd.

Region 2

Goal: Maximize power in below-rated operation

- A theoretical constant can be used that provides “optimal” generator torque for a given rotor speed.
 - This is subject to modeling errors and the generator torque dynamics are not (generally) tuned
 - Known as the “ $K\omega^2$ ” law
- Variable Speed wind turbines are designed for an optimal aerodynamic efficiency at a given tip-speed ratio.
 - The generator torque controller can be designed to track this TSR

Theory – The standard controller contd.

Region 3

Goal: Regulate rotor speed

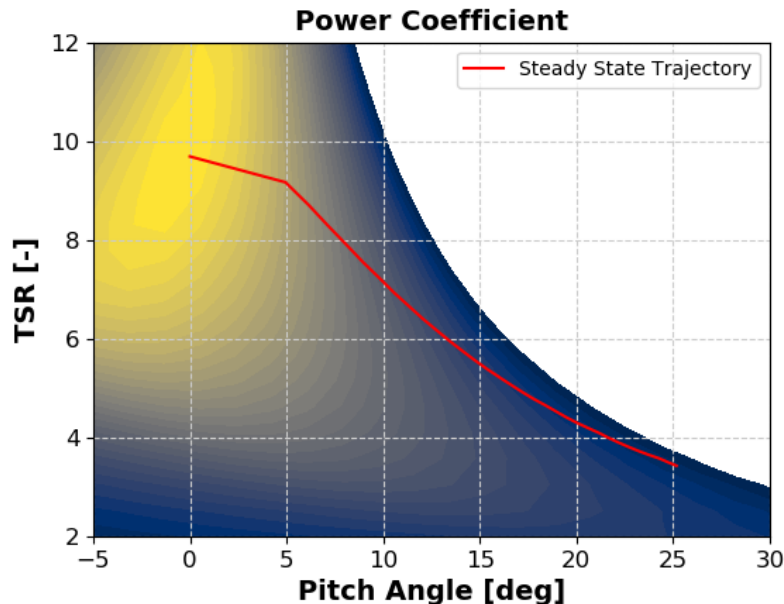
- Almost always a “gain scheduled” controller to track a reference rotor speed
 - Aerodynamic sensitivities are much lower near-rated than above-rated
- Tuning of this controller can be difficult without reasonable models of these sensitivities

Theory – Coefficient surfaces

Turbine coefficient look-up tables

CCBlade is used to generate power, thrust, and torque coefficient tables

- Steady state “trajectory” of expected wind turbine operation used for gain scheduling
- Used to determine system parameters and controller setpoints
- Power coefficient is used by wind speed estimators in ROSCO



Theory – ROSCO's Additional Functionalities

Other Controller functions

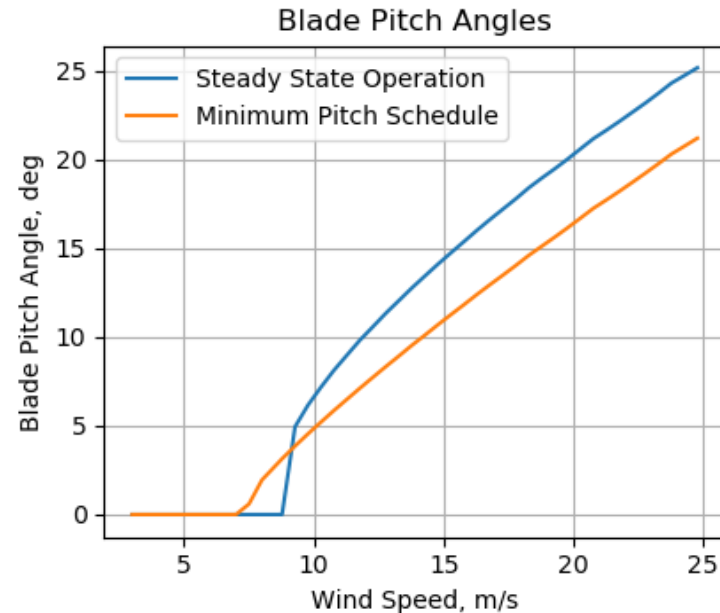
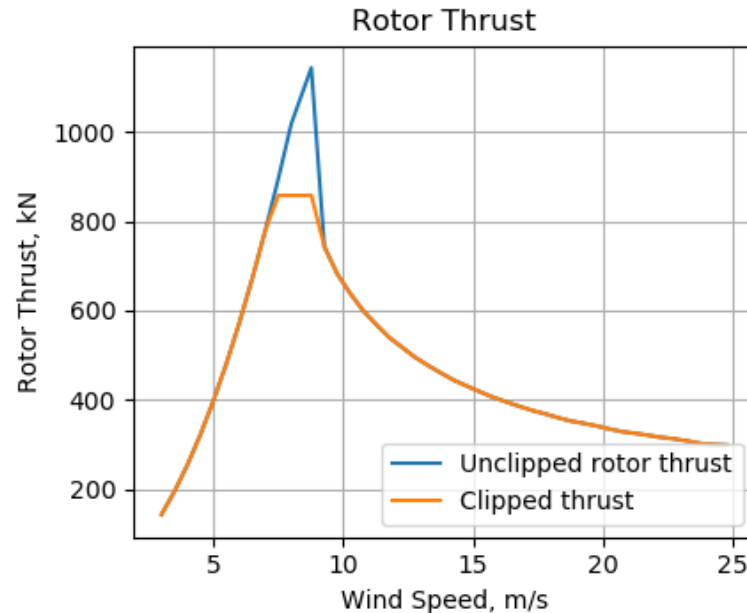
ROSCO has additional or updated capabilities that are consistent with industry standards and improve on previous reference controllers

- Smoothed transition region
- Wind Speed Estimators
- Pitch saturation
- Shutdown procedures
- Individual Pitch Control
- Yaw control
- Floating offshore wind turbine control
- Distributed aerodynamic control

Theory – Peak Shaving

Peak shaving

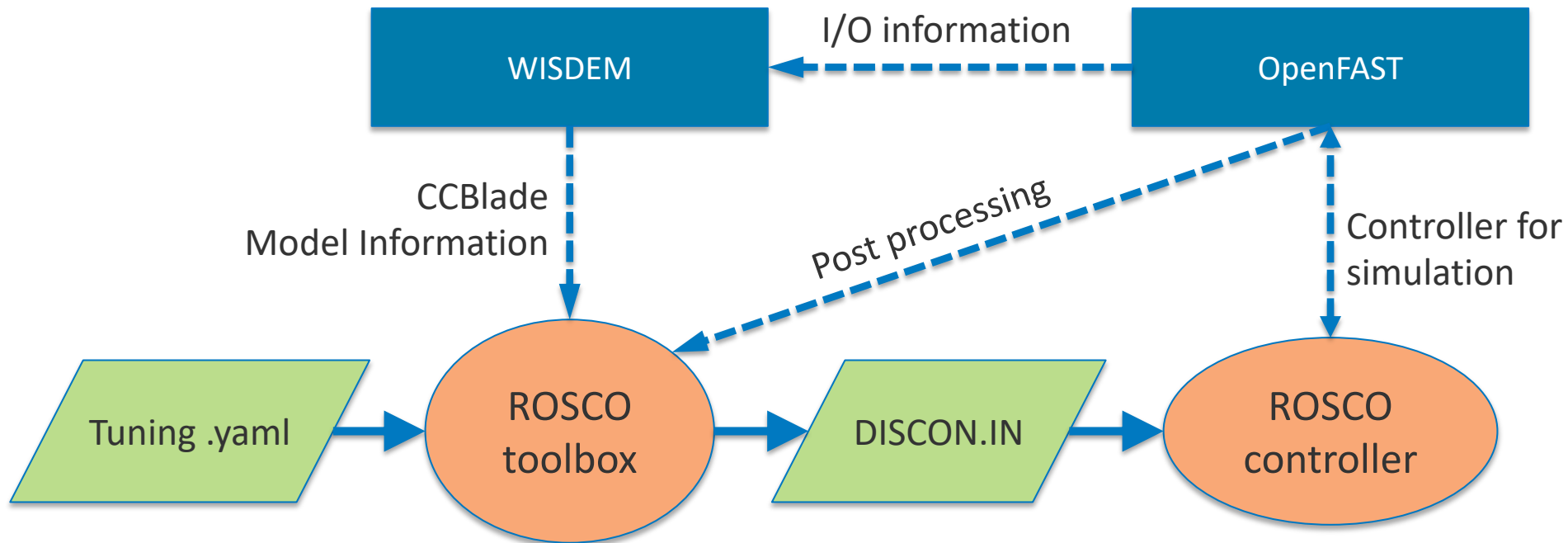
Peak shaving can be used to reduce tower loads by reducing rotor thrust near rated. In this tutorial, we will show how to implement it.



Implementation Overview

How does all of this work?

Implementation - ROSCO Toolchain Workflow



* Note: This flow chart is not representative of an optimization procedure using WISDEM and OpenFAST

Implementation - .yaml file

.yaml files

We use yaml files as the top-level input file to the ROSCO generic tuning process.

Three main parts:

- Path Parameters
 - Contains the path details of the OpenFAST input files
- Turbine Parameters
 - Contains some high level turbine parameters. Rated rotor speeds, relevant wind speeds, rated power, etc...
 - Generally known a-priori
- Controller Parameters
 - Contains the controller tuning parameters:
 - Flags to define controller types
 - Four necessary tuning parameters
 - Optional parameters for fine tuning

Implementation - .yaml path parameters


.yaml – path parameters

path_params:

FAST_InputFile: 'OpenFAST_BAR_00.fst'

FAST_directory: './ROSCO_tutorial/BAR_00/'


rotor_performance_filename: 'Cp_Ct_Cq_BAR_00.txt'

- 
- Optional text file containing C_p , C_t , C_q surfaces
 - Without text file, CCBlade is run to calculate surfaces

Implementation - .yaml turbine parameters

.yaml – turbine parameters

```
turbine_params:
  rotor_inertia: 297997280.000 # Rotor inertia [kg m^2]
  rated_rotor_speed: 0.801147314 # Rated rotor speed [rad/s]
  v_min: 3.0 # Cut-in wind speed [m/s]
  vRated: 8.27356 # Rated wind speed [m/s]
  v_max: 25.0 # Cut-out wind speed [m/s]
  max_pitch_rate: 2. # Maximum blade pitch rate [deg/s]
  max_torque_rate: 4500000. # Maximum torque rate [Nm/s]
  rated_power: 5000000. # Rated Power [W]
  bld_edgewise_freq: 4.31752 # Blade edgewise first natural frequency [rad/s]
  bld_flapwise_freq: 2.7209 # Blade flapwise first natural frequency [rad/s]
  TSR_operational: # None # Desired below-rated operational tip speed ratio
```

- 
- Tuning process will use the optimal value from the C_p -surface if this is not defined

Implementation - .yaml controller parameters

.yaml – control parameter flags

```
controller_params:
  # Flags
  LoggingLevel: 1          # Debug level
  F_LPFType: 2             # Low-pass Filter type
  F_NotchType: 0           # Notch filter on/off
  IPC_ControlMode: 0       # Individual pitch control type (1P, 2P, 1P+2P)
  VS_ControlMode: 2        # Generator Torque Control Type
  PC_ControlMode: 1        # Pitch control on/off
  Y_ControlMode: 0         # Yaw control type
  SS_Mode: 1              # Setpoint Smoother on/off
  WE_Mode: 2              # Wind Speed Estimator type
  PS_Mode: 0              # Pitch saturation type
  SD_Mode: 0              # Shutdown on/off
  Fl_Mode: 0              # Floating control on/off
  Flp_Mode: 0             # Distributed aerodynamic (flap) control type
```

* These should not generally need to be changed, unless different control logic is desired

Implementation - .yaml controller parameters

.yaml – control parameter flags

```
controller_params:
  # Tuning parameters
  zeta_pc: 0.8          # Pitch controller desired damping ratio [-]
  omega_pc: 0.2         # Pitch controller desired natural frequency [rad/s]
  zeta_vs: 0.55        # Torque controller desired damping ratio [-]
  omega_vs: 0.1        # Torque controller desired natural frequency [rad/s]
```

Some rules of thumb

These are the only four necessary “tuning parameters” for the standard controller implementation. If `VS_ControlMode = 0/1`, only `zeta_pc` and `omega_pc` are needed. For TSR tracking torque controller, `zeta_vs` and `omega_vs` are needed.

- larger `omega` = faster pitch/torque response, too large = unstable
- larger `zeta` = smoother rotor speed signals

	<code>zeta_pc</code>	<code>omega_pc</code>	<code>zeta_vs</code>	<code>omega_vs</code>
Large turbines (>75m blades)	0.7 - 1.3	0.15 - 0.3	0.8 - 1.3	0.1 – 0.4
“Small” turbines (<75m blades)	0.5 - 1.0	0.5 - 0.8	0.5 – 1.0	0.1 - 0.4

Implementation – Tuning Script

A typical tuning script will:

1. Load an ROSCO tuning .yaml file
2. Initialize the ROSCO toolbox turbine, controller, and utilities classes
3. Load a turbine model
4. Run the tuning processes
5. Write DISCON.IN for use by the controller

An example tuning script is provided in:

```
<tutorial_dir>/ROSCO_tutorial/run_ROSCOtuning.py
```

Implementation – DISCON.IN

The DISCON.IN file is used by the controller

The ROSCO toolbox tuning scripts will write DISCON.IN, an input file for the controller.

DISCON.IN, along with the controller binary, is pointed to in ServoDyn using the `DLL_InFile` and `DLL_FileName` parameters.

Once you compile ROSCO, you do not need to re-compile it again (unless, of course, there are updates that you want). You only need one controller binary, and can simply modify DISCON.IN for your wind turbines.

DISCON.IN has ~100 input parameters. Some are simply the controller flags, the rest define the controller behavior (gains, setpoints, etc...).

Using the ROSCO toolbox

Lets tune a controller

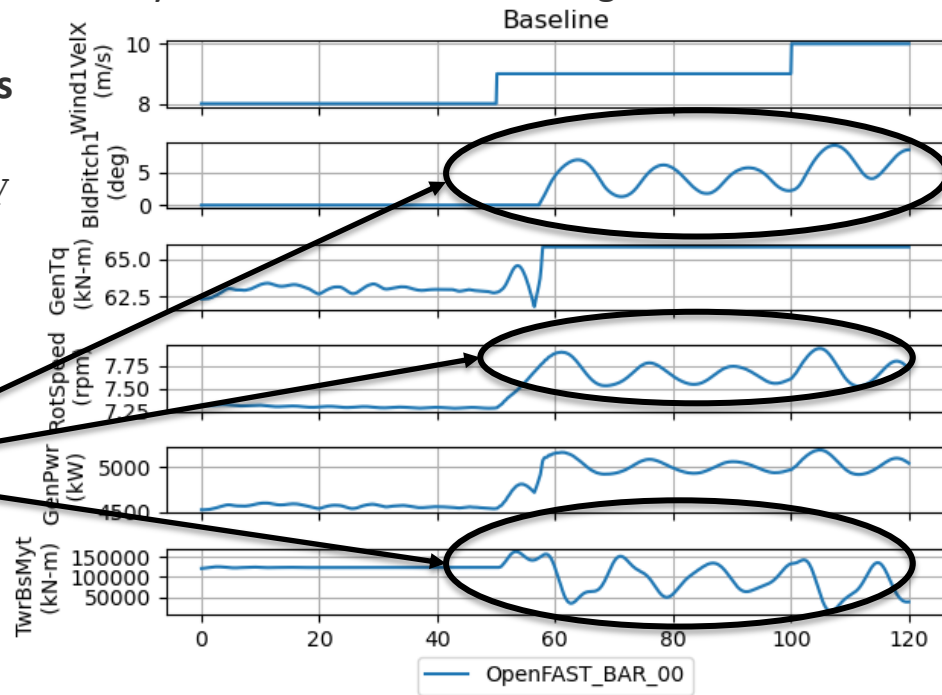
ROSCO toolbox – Initial tuning

We will start by just running the distributed files and plotting the results. The curious user can look into `run_ROSCOtuning.py` and see the necessary commands for the tuning.

Run the turbine model and plot some results

```
python run_ROSCOtuning.py
python plot ROSCOresults.py
```

All indicative of poor pitch controller tuning



ROSCO toolbox – Improve the controller

Improve the response, reduce loads

The blade pitch controller is too aggressive, so there are fairly significant rotor speed and blade pitch fluctuations during the step transitions in wind speed. Lets decrease `omega_pc` for a slower blade pitch response (but not too much!). We also think we might want peak shaving to reduce tower loads.

1. Open `<tutorial_dir>/ROSCO_tutorial/BAR_00/ServoData/BAR_00.yaml`
2. Decrease `omega_pc` in the `controller_params` section to `0.2`
3. Change `PS_Mode` in the `controller_params` section to `1`

Before we re-run, lets rename our previous output file so we can still plot it:

On Mac/Linux:

```
cp BAR_00/OpenFAST_BAR_00.outb BAR_00/OpenFAST_BAR_00_run1.outb
```

On Windows

```
copy BAR_00\OpenFAST_BAR_00.outb BAR_00\OpenFAST_BAR_00_run1.outb
```

ROSCO toolbox – Improve the controller

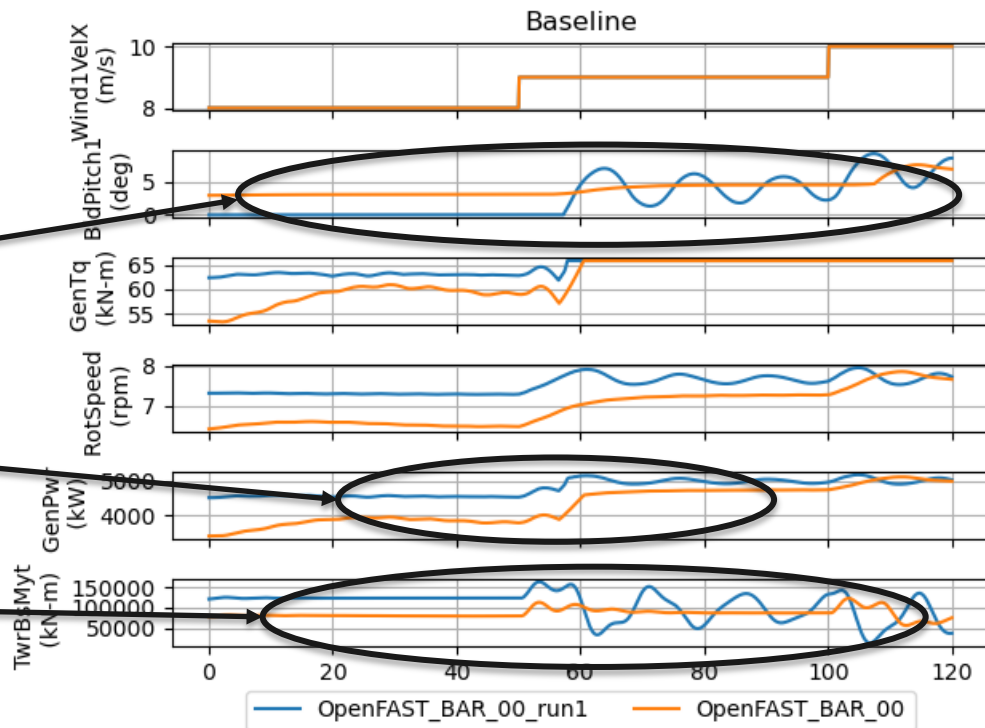
Re-run and plot

```
python run_ROSCOtuning.py  
python plot_ROSCoresults.py
```

Saturated blade pitch
and reduced fluctuations

Power Losses

Reduced Tower Bending



ROSCO toolbox – add peak shaving

Remove peak shaving

This looks better, but maybe we aren't so concerned with tower-base loads after all. We can just turn this off without re-tuning entirely

1. **Open** `<tutorial_dir>/ROSCO_tutorial/BAR_00/OpenFAST_BAR_00_DISCON.IN`
2. **Change** `PS_Mode` in the **Controller Flags** section to 0

Lets rename our previous output file again:

On Mac/Linux:

```
cp BAR_00/OpenFAST_BAR_00.outb BAR_00/OpenFAST_BAR_00_run2.outb
```

On Windows

```
copy BAR_00\OpenFAST_BAR_00.outb BAR_00\OpenFAST_BAR_00_run2.outb
```

ROSCO toolbox – Improve the controller

Re-run just OpenFAST and plot

```
cd BAR_00
```

On mac/linux

```
openfast OpenFAST_BAR_00.fst
```

On Windows (32bit)

```
..\OpenFAST_Executables\openfast_Win32.exe OpenFAST_BAR_00.fst
```

On Windows (63bit)

```
..\OpenFAST_Executables\openfast_x64.exe OpenFAST_BAR_00.fst
```

```
cd ..
```

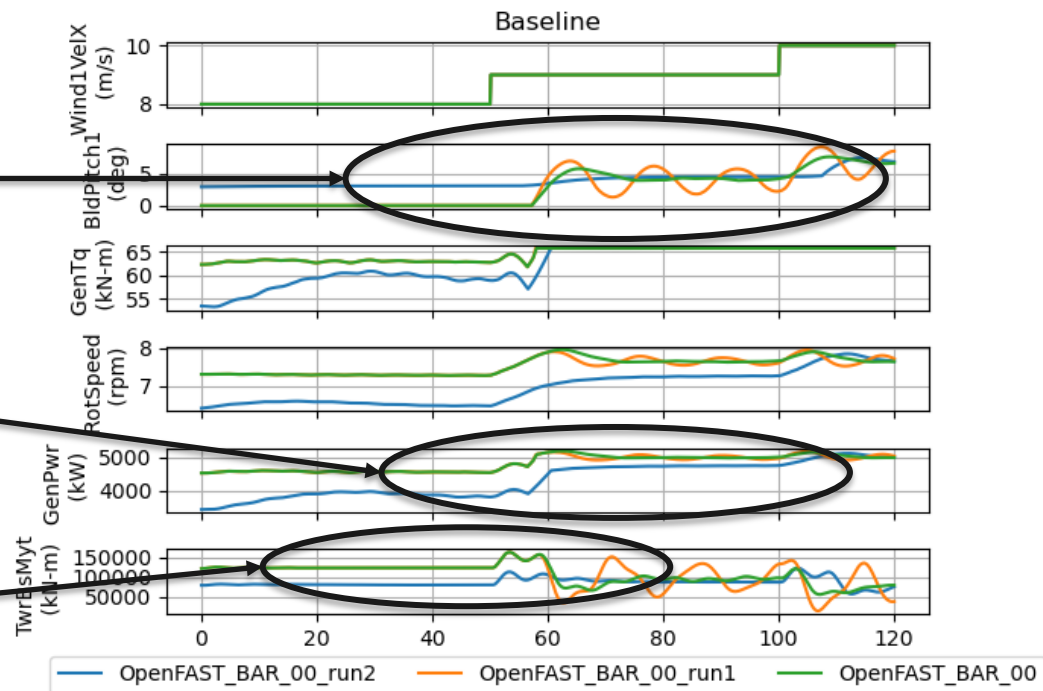
```
python plot_ROSCOresults.py
```

ROSCO toolbox – Improve the controller

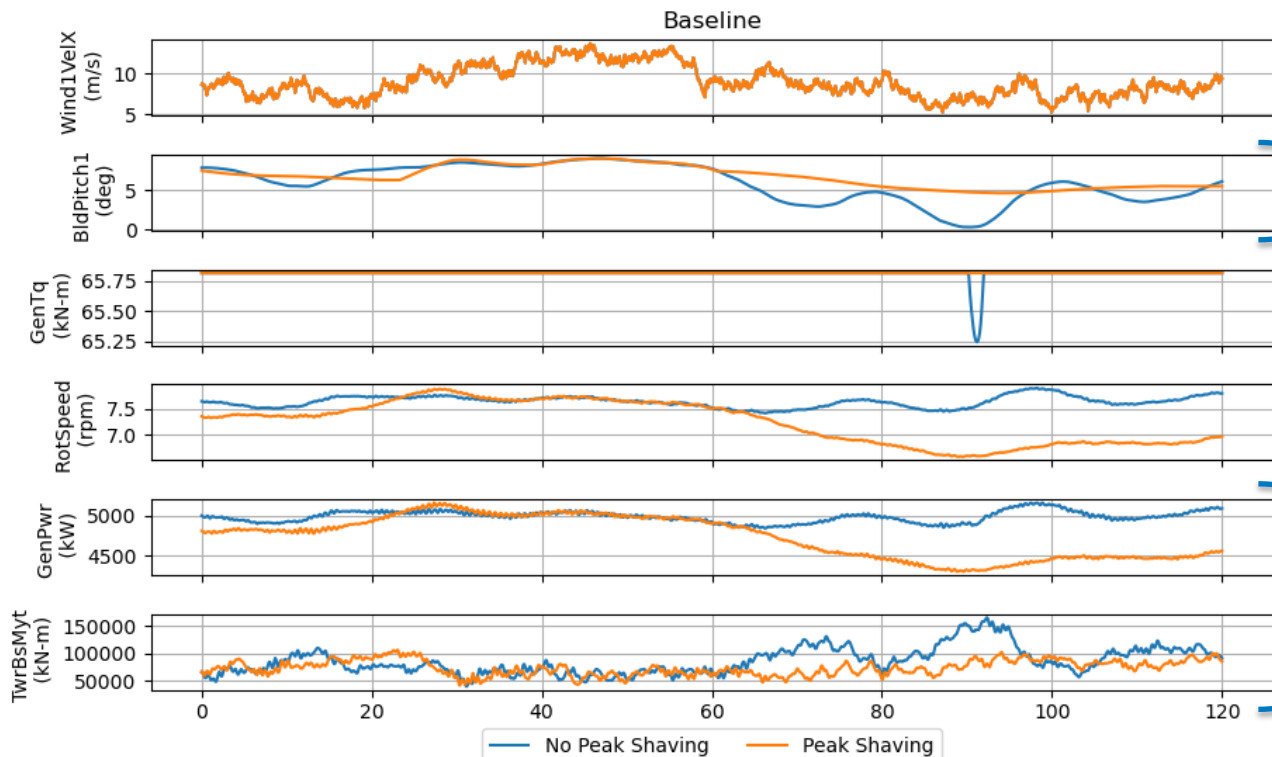
Smoother blade pitch response,
but without saturation

No Power Losses

No longer have reduced
tower bending moments



Turbulent Results – Peak shaving



- Pitch saturation near rated

- Reduced Tower Base loads
- Power loss as a consequence

Future Work

Acknowledgements

Resources

What's to come?

Future work

Control Modules

More controller modules and capabilities to come!

- Power reference tracking control
- Improved yaw control
- Expanded shutdown logic
- Tower resonance avoidance
- Improved FOWT control
- Auto-tune IPC

Optimization

Controller development is becoming increasingly integrated into optimization tools based at NREL

- Stronger integration into WISDEM & WEIS
- Controller optimization using OpenFAST
- Wind turbine co-design optimization using ROSCO

Acknowledgments & Resources

Special thanks to all who have contributed and helped with the ROSCO development process:

- Jan-Willem van Wingerden and Sebastiaan Mulders at TU Delft
- Paul Fleming
- Rafael Mudafort
- Pietro Bortolotti, Evan Gaertner, and others on the system's engineering team
- Everyone else who has provided feedback and dug out the bugs!

Initial publication on this work:

<https://doi.org/10.1088/1742-6596/1452/1/012010>

NREL publications:

<https://doi.org/10.1088/1742-6596/1452/1/012002>

...more detailed publications to come!

Thank you!

www.nrel.gov

This work was authored by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Funding provided by the U.S. Department of Energy Office of Energy Efficiency and Renewable Energy Wind Energy Technologies Office. The views expressed in the article do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.

