NREL
NATIONAL RENEWABLE ENERGY LABORATORY

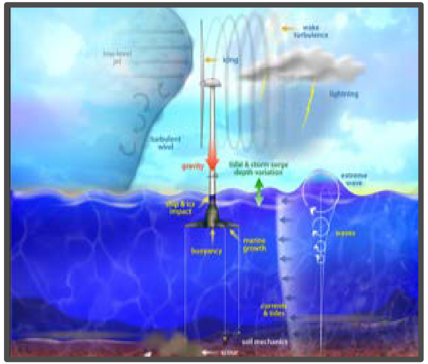
Starting with WISDEM™
The Short Course

Garrett Barter

5th Wind Energy Systems Engineering Workshop (WESE 2019)
Pamplona, Spain, October 2-4, 2019

NREL has both turbine- and plant/system-focused research programs

Turbine Focused



OpenFAST

Computer engineering tool for simulating the coupled dynamic response of wind turbines

Flagship software product

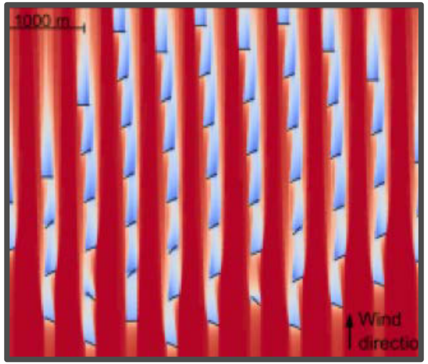
System Focused

WISDEM™

Wind-plant Integrated System Design and Engineering Model

FOCUS

Farm Focused



FLORIS

A controls-oriented engineering wake model

FAST.Farm

WindSE

NALU

FOCUS

Agenda

- (Brief) Introduction to WISDEM and OpenMDAO
- WISDEM installation (essentials only)
- Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks
- Some more detail on OpenMDAO
- Tutorial 2: Finding the Betz Limit through OpenMDAO optimization
- ~~Tutorial 3: Sellar problem for putting multiple models together~~
- Tutorial 4: Modeling a whole turbine and plant LCOE

(Brief) Introduction to WISDEM and OpenMDAO

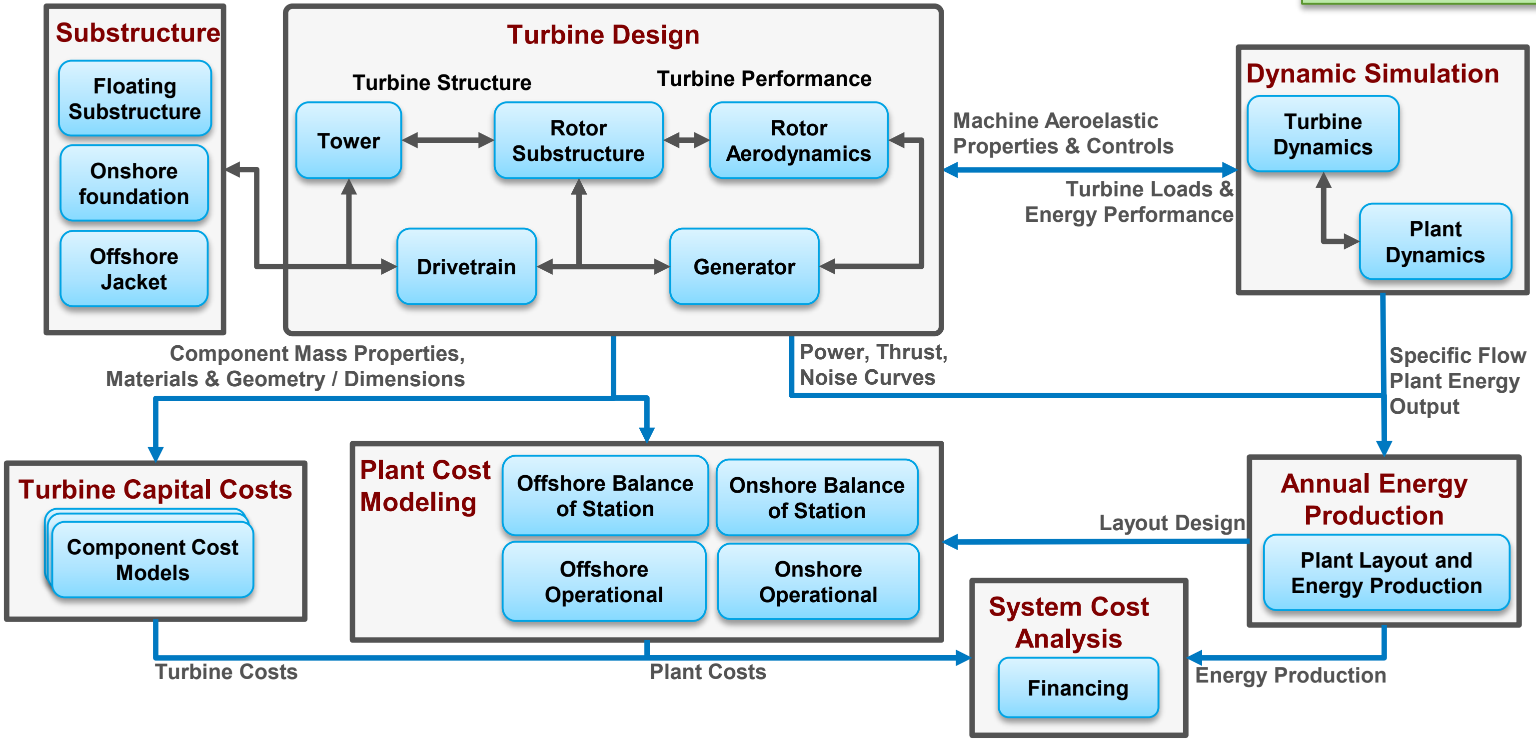
- **(Brief) Introduction to WISDEM and OpenMDAO**
- WISDEM installation (essentials only)
- Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks
- Some more detail on OpenMDAO
- Tutorial 2: Finding the Betz Limit through OpenMDAO optimization
- Tutorial 4: Modeling a whole turbine and plant

WISDEM: Creates a virtual, vertically integrated wind plant from components to operations

WISDEM: Wind-Plant Integrated System Design & Engineering Model

- Integrated turbine design (e.g. rotor aero-structure, full turbine optimization)
- Integrated plant design and operations (e.g. wind plant controls and layout optimization)
- Integrated turbine and plant optimization (e.g. site-specific turbine design)

Modular design allows "plug-and-play" with external (3rd party) component modules



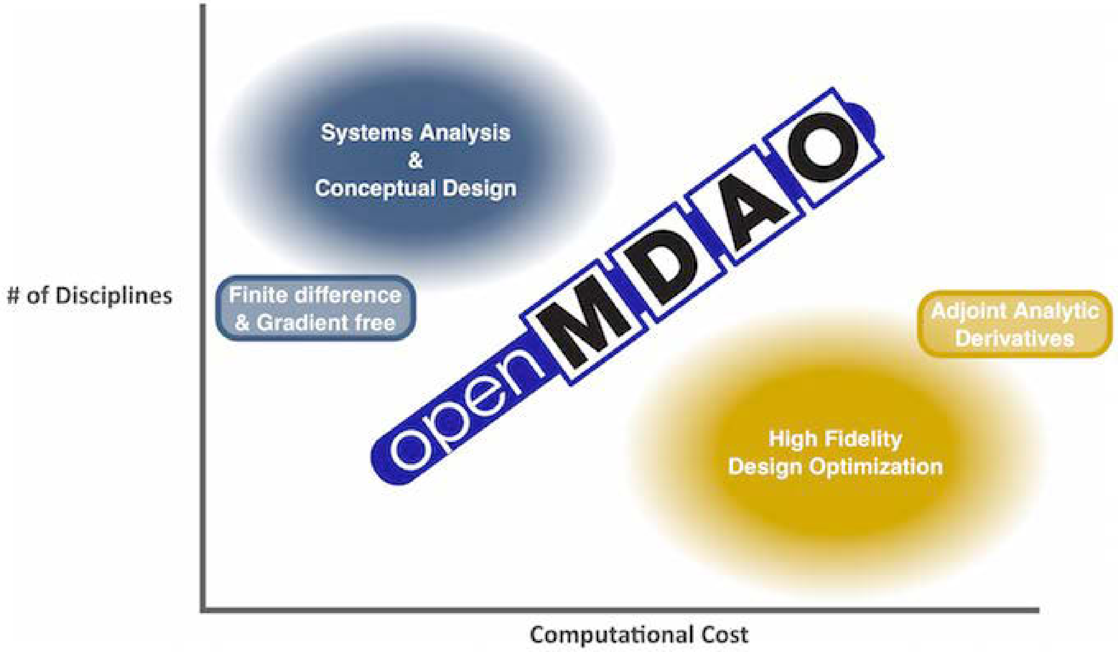
Software platform is built with Python using the OpenMDAO library

- **Most WISDEM modules are developed in Python using the OpenMDAO library**
  - Underlying analysis may be in C, C++ or Fortran

OpenMDAO (openmdao.org)

- **Open-source, python-based platform for systems analysis and multidisciplinary optimization**
- **Provides ”glue code” and “drivers/wrappers”**
- **Enables**
  - Model decomposition
  - Ease of development and maintenance
  - Tightly coupled solutions and parallel methods

Phython



WISDEM installation (essentials only)

- (Brief) Introduction to WISDEM and OpenMDAO
- **WISDEM installation (essentials only)**
- Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks
- Some more detail on OpenMDAO
- Tutorial 2: Finding the Betz Limit through OpenMDAO optimization
- Tutorial 4: Modeling a whole turbine and plant

WISDEM install (essentials only): For full instructions see nwtc.nrel.gov/wisdem

- **Key steps**
  1. Download and install Anaconda3 64-bit from https://www.anaconda.com/distribution
  2. Setup new "conda environment" (provides digital sandbox to explore WISDEM without impacting any other part of your system)
  3. Install WISDEM and its dependencies
  4. Download WISDEM source code from GitHub

- **We want to install the code like a simple user but take a peek at the files like a developer**
- **Open the Anaconda Power Shell (Windows) or Terminal App (Mac) and do:**

```
conda config --add channels conda-forge
conda create -y --name wisdem-env python=3.7
conda activate wisdem-env
conda install –y wisdem git jupyter
git clone https://github.com/WISDEM/WISDEM.git
```

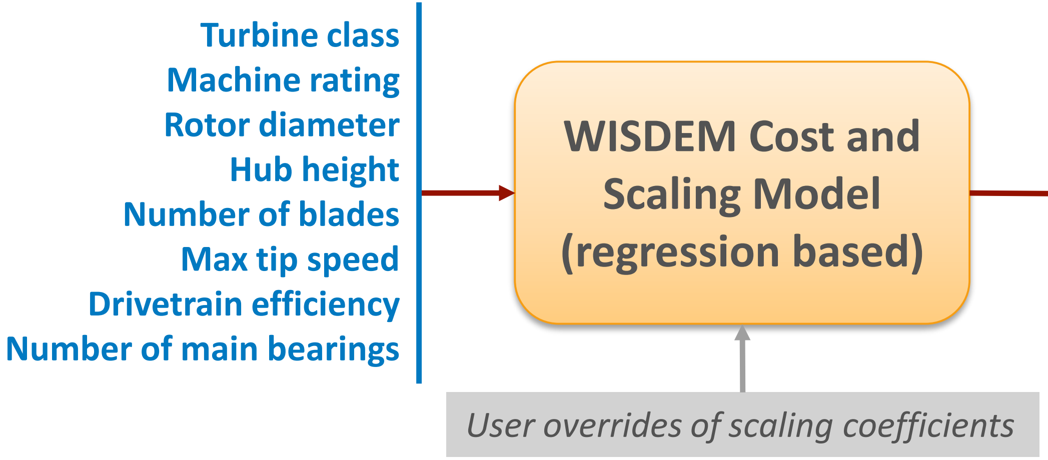
Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks

- (Brief) Introduction to WISDEM and OpenMDAO
- WISDEM installation (essentials only)
- **Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks**
- Some more detail on OpenMDAO
- Tutorial 2: Deriving the Betz Limit through OpenMDAO optimization
- Tutorial 4: Modeling a whole turbine and plant

Use WISDEM as a calculator to estimate component masses and costs from simple scaling relationships

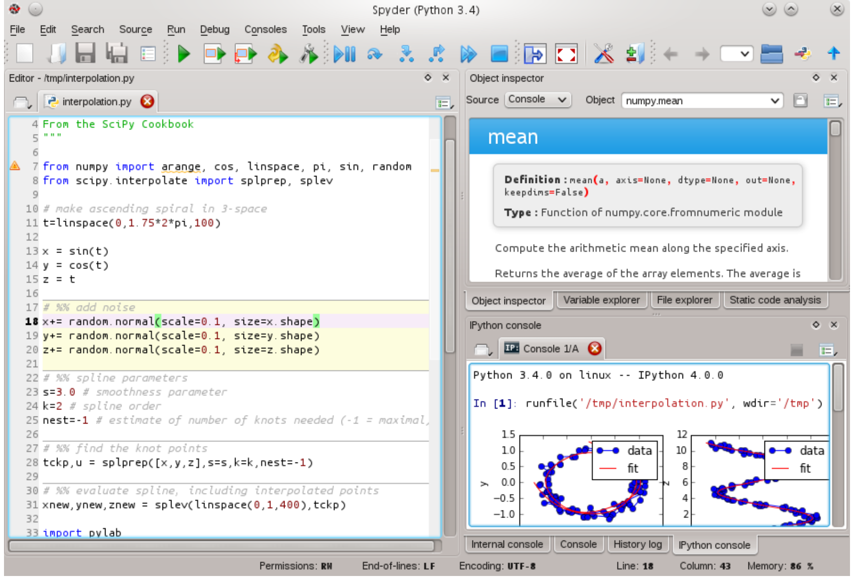
- **WISDEM has multiple levels of fidelity, we will operate at the simplest level: "spreadsheet"-type calculation of component masses and cost**

- **We will: Populate inputs, execute model, list all the model inputs and outputs**
  - Will reveal some of the backend layers of OpenMDAO building blocks
  - Will ignore OpenMDAO syntax for now

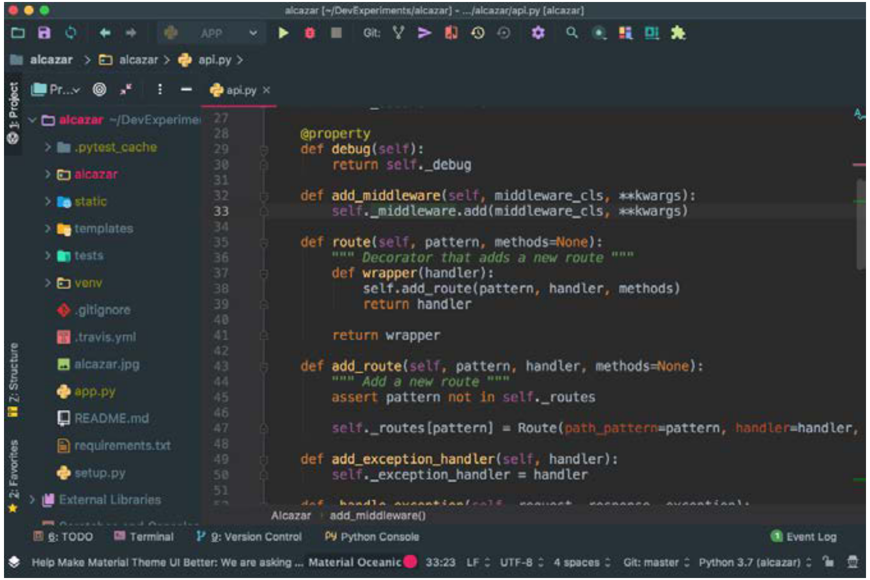


There are many ways to run WISDEM (and python) beyond Jupyter Notebooks

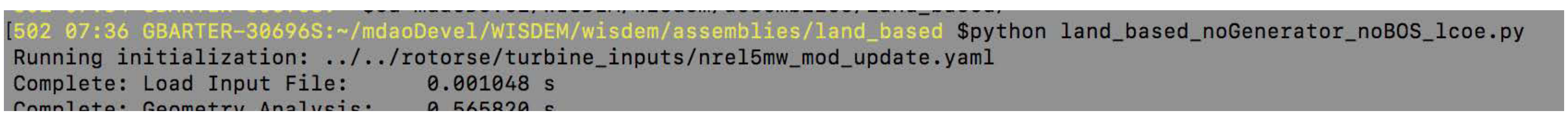
Spyder for a Matlab-style Desktop



PyCharm or other IDE



Command line from Anaconda Prompt or Terminal App



Jupyter Notebook: Interacting with the Python “shell” through a browser in a live code “diary”

- **Jupyter Notebook is a web application that connects with your local python shell**

- **Allows for creating and sharing documents with**
  - Live code
  - Equations
  - Visualizations
  - Narrative text

- **To get started with the WISDEM Jupyter Notebook tutorials we have to navigate to the right directory and start Jupyter**
- **Open the Anaconda Prompt (Windows) or Terminal App (Mac) and do:**

```
cd WISDEM/tutorial-notebooks
jupyter notebook
```

gbarter@GBARTER-30696S: ~/mdaoDevel/WISDEM/tutorial-notebooks — jupyter-notebook — 105×27

```
(wisdem-env) 518 22:23 GBARTER-30696S:~/mdaoDevel $cd WISDEM/tutorial-notebooks/
(wisdem-env) 519 22:23 GBARTER-30696S:~/mdaoDevel/WISDEM/tutorial-notebooks $jupyter notebook
[I 22:23:18.211 NotebookApp] Serving notebooks from local directory: /Users/gbarter/mdaoDevel/WISDEM/tuto
rial-notebooks
[I 22:23:18.211 NotebookApp] The Jupyter Notebook is running at:
[I 22:23:18.211 NotebookApp] http://localhost:8888/?token=5b482388f789a2bc572b8dc0ed2feea042221b3a2057abf
d
[I 22:23:18.211 NotebookApp]  or http://127.0.0.1:8888/?token=5b482388f789a2bc572b8dc0ed2feea042221b3a205
7abfd
[I 22:23:18.212 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip c
onfirmation).
[C 22:23:18.296 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/gbarter/Library/Jupyter/runtime/nbserver-6182-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=5b482388f789a2bc572b8dc0ed2feea042221b3a2057abfd
     or http://127.0.0.1:8888/?token=5b482388f789a2bc572b8dc0ed2feea042221b3a2057abfd
```


jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/
Name
Last Modified
File size
img
5 days ago
01_cost_and_scaling.ipynb
5 days ago
63.4 kB
02_betz_limit.ipynb
5 days ago
18.1 kB
03_sellar.ipynb
5 days ago
18.7 kB
04_turbine_assembly.ipynb
5 days ago
35.2 kB
WISDEM_notebooks.pdf
5 days ago
804 kB

Use Shift+Enter to evaluate each block

Tutorial 1. Cost and scaling model (for a turbine)

WISDEM offers different levels of fidelity for estimating mass, cost, and LCOE. The simplest form is an implementation of a "spreadsheet" type model, where given a few high-level parameters, such as `rotor diameter` and `machine_rating`, the user can use a series of empirical relationships (regression-based) to estimate turbine mass and cost. This is called the Cost and Scaling Model and we will call it here.

First, import the WISDEM code that will model our costs. Also import the pieces of OpenMDAO that we'll need for the calculation.

In [1]:
```
from wisdem.turbine_costsse.nrel_csm_tcc_2015 import nrel_csm_2015
from openmdao.api import Problem
import numpy as np
```

Cost and Mass model

`nrel_csm_2015` is a class that contains a model to estimate the masses and costs of turbine components.

First, we instantiate the `nrel_csm_2015` class. Then we make an OpenMDAO `Problem` that uses this class. After that, we set the inputs for our model.

In [2]:
```
turbine = nrel_csm_2015()
prob = Problem(turbine)
prob.setup()

prob['rotor_diameter'] = 126.0
prob['turbine_class'] = 1
prob['blade_has_carbon'] = False
prob['blade_number'] = 3
prob['machine_rating'] = 5000.0
prob['hub_height'] = 90.0
prob['bearing_number'] = 2
prob['crane'] = True
prob['max_tip_speed'] = 80.0
prob['max_efficiency'] = 0.90
```

Now we are set to run the model:

Now we are set to run the model:

In [11]:
```
prob.run_driver()
```

Out[11]:
```
False
```

That's it! The model has been executed and now we have to get at the outputs we are interested in. Since we don't know the names of the outputs variables, we can just exhaustively list (and store) all of the model inputs and outputs. There are simple commands for doing so:

In [4]:
```
myinputs = prob.model.list_inputs(units=True)
```

```
219 Input(s) in 'model'
-----------------------

varname                                     value               units
------------------------------------------- ------------------  ------
top
  nrel_csm_mass
    blade
      rotor_diameter                        [126.]              m
      blade_mass_coeff                      [0.5]               None
      blade_user_exp                        [2.5]               None
    hub
      blade_mass                            [18590.66820649]    kg
      hub_mass_coeff                        [2.3]               None
      hub_mass_intercept                    [1320.]             None
    pitch
      blade_mass                            [18590.66820649]    kg
      pitch_bearing_mass_coeff              [0.1295]            None
      pitch_bearing_mass_intercept          [491.31]            None
```

Now list the model outputs.

In [5]:
```
myoutputs = prob.model.list_outputs(units=True)
```

```
91 Explicit Output(s) in 'model'
```

User Overrides

The Cost and Scaling model can be used to estimate mass and cost from a limited set of inputs. If the user already knows the mass or cost of a particular component, the easiest thing to do is to override the mass or cost scaling coefficient. This can also be used to conduct scaling studies at different and mass and cost growth sensitivities:

In [6]:
```python
prob['gearbox_mass_coeff'] = 75.0
prob['gearbox_mass_cost_coeff'] = 10.0
prob.run_driver()
```

Out[6]:
```
False
```

If we store these inputs and outputs into new containers, we can easily compare the impact of the changes

In [7]:
```python
newinputs = prob.model.list_inputs(units=True)
```

```
219 Input(s) in 'model'
-----------------------

varname                                      value               units
-------------------------------------------  ------------------  ------
top
  nrel_csm_mass
    blade
      rotor_diameter                         [126.]              m
      blade_mass_coeff                       [0.5]               None
      blade_user_exp                         [2.5]               None
    hub
      blade_mass                             [18590.66820649]    kg
      hub_mass_coeff                         [2.3]               None
      hub_mass_intercept                     [1320.]             None
    pitch
      blade_mass                             [18590.66820649]    kg
      pitch_bearing_mass_coeff               [0.1295]            None
      pitch_bearing_mass_intercept           [491.31]            None
```

In [8]:
```python
newoutputs = prob.model.list_outputs(units=True)
```

```
In [8]: newoutputs = prob.model.list_outputs(units=True)
```

```
91 Explicit Output(s) in 'model'
-------------------------------

varname                                      value               units
-------------------------------------------  ------------------  ------
top
  nrel_csm_mass
    sharedIndeps
      machine_rating                         [5000.]             kW
      rotor_diameter                         [126.]              m
    blade
      blade_mass                             [18590.66820649]    kg
    hub
      hub_mass                               [44078.53687493]    kg
    pitch
      pitch_system_mass                      [10798.90594644]    kg
    spinner
      spinner_mass                           [973.]              kg
    lss
```

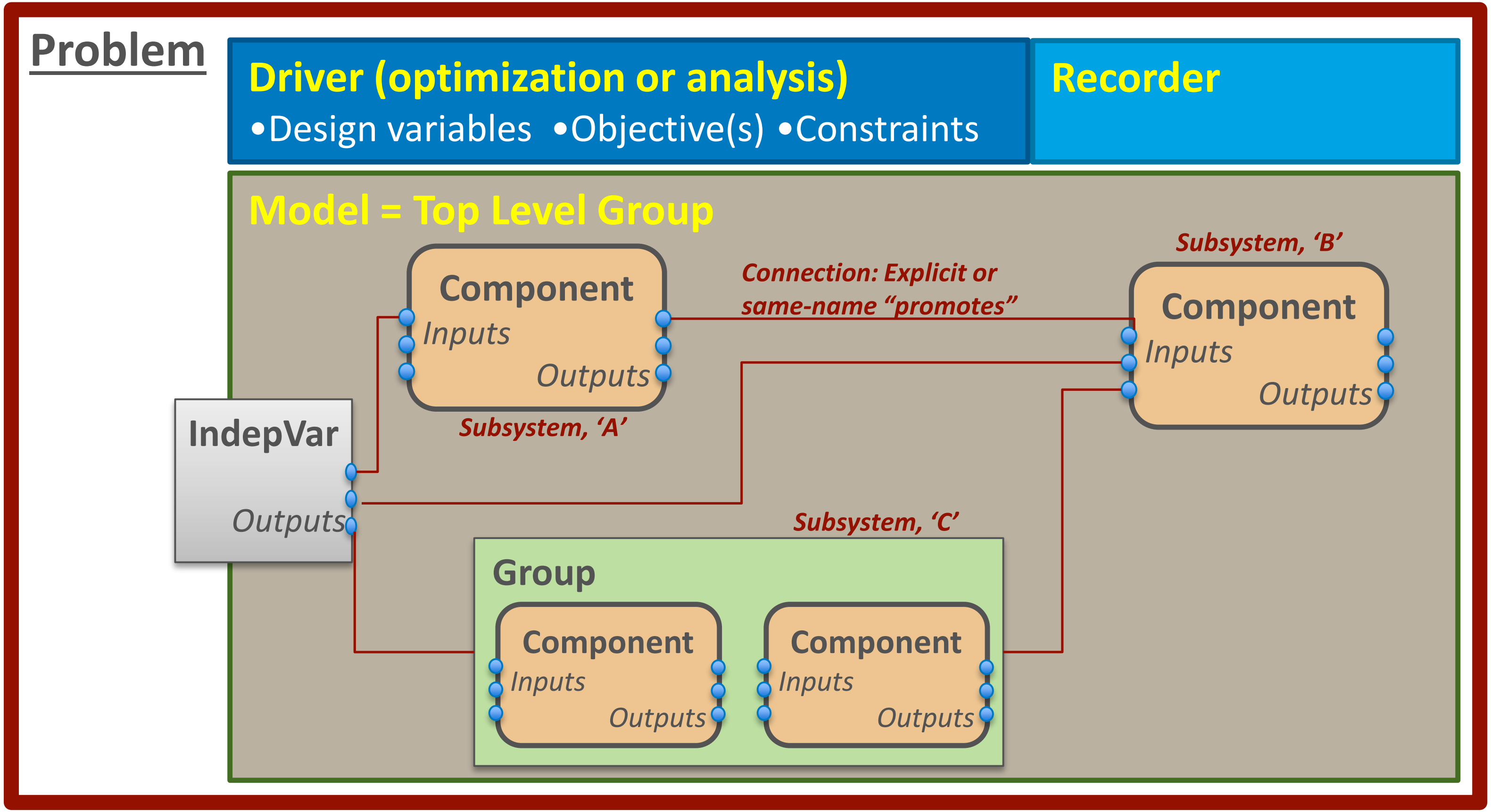
Scrolling through the outputs, we can see that these new values for gearbox mass and cost changed the cost of the turbine from $726/kW to $672/kW.

Another approach is to split out the mass scaling and cost scaling routines, but we will leave that for another tutorial

Some more detail on OpenMDAO

- (Brief) Introduction to WISDEM and OpenMDAO
- WISDEM installation (essentials only)
- Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks
- **Some more detail on OpenMDAO**
- Tutorial 2: Deriving the Betz Limit through OpenMDAO optimization
- Tutorial 4: Modeling a whole turbine and plant

OpenMDAO building blocks and concepts



Tutorial 2: Finding the Betz Limit through OpenMDAO optimization

- (Brief) Introduction to WISDEM and OpenMDAO
- WISDEM installation (essentials only)
- Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks
- Some more detail on OpenMDAO
- Tutorial 2: Finding the Betz Limit through OpenMDAO optimization
- Tutorial 4: Modeling a whole turbine and plant

OpenMDAO model building steps applied to the Betz Problem

Component Steps

- **Create an OpenMDAO *Component***
- **Add the actuator disk inputs and outputs**
- **Use *declare_partials()* to declare derivatives**
  - Finite difference or exact analytic options are available
- **Create a *compute()* method to compute outputs from inputs**
- **Create a *compute_partials()* method for the derivatives**

Group and Problem Steps

- **Create an OpenMDAO *Group*.**
  - Add a subsystem of independent variables
  - Add the disk Component as a subsystem
  - Connect variables through *connect()* statements or same name promotion
- **Create an OpenMDAO *Problem***
  - Set the model = Group instance
  - Add optimization *Driver*
  - Add design variables
  - Add the objective
- **Setup and run problem driver**


jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/
Name
Last Modified
File size
img 5 days ago
01_cost_and_scaling.ipynb 5 days ago 63.4 kB
02_betz_limit.ipynb 5 days ago 18.1 kB
03_sellar.ipynb 5 days ago 18.7 kB
04_turbine_assembly.ipynb 5 days ago 35.2 kB
WISDEM_notebooks.pdf 5 days ago 804 kB

Tutorial 2: Betz Limit

Now that we have ran a simple calculator model using WISDEM, let's look at OpenMDAO. OpenMDAO is the code that connects the various components of turbine models into a cohesive whole that can be optimized in systems engineering problems. WISDEM uses OpenMDAO to build up modular *components* and *groups* of components to represent a wind turbine. Fortunately, OpenMDAO already provides some excellenet training examples on their website. This tutorial is based on the OpenMDAO example, Optimizing an Actuator Disk Model to Find Betz Limit for Wind Turbines, which we have extracted and added some additional commentary. The aim of this tutorial is to summarize the key points you'll use to create basic WISDEM models. For those interested in WISDEM development, getting comfortable with all of the core OpenMDAO training examples is strongly encouraged.

A classic problem of wind energy engineering is the Betz limit. This is the theoretical upper limit of power that can be extracted from wind by an idealized rotor. While a full explanation is beyond the scope of this tutorial, it is explained in many places online and in textbooks. One such explanation is on Wikipedia https://en.wikipedia.org/wiki/Betz%27s_law .

Problem formulation

According to the Betz limit, the maximum power a turbine can extract from wind is:

$$C_p = \frac{16}{27} \approx 0.593$$

Where C_p is the coefficient of power.

We will compute this limit using OpenMDAO by optimizing the power coefficient as a function of the induction factor (ratio of rotor plane velocity to freestream velocity) and a model of an idealized rotor using an actuator disk.

Here is our actuator disc:

OpenMDAO implementation

First we need to import OpenMDAO

In [1]:
```
import openmdao.api as om
```

Now we can make an `ActuatorDisc` class that models the actuator disc for the optimization.

In [2]:
```
class ActuatorDisc(om.ExplicitComponent):
    def setup(self):
        # Inputs into the the model
        self.add_input('a', 0.5, desc='Indcued velocity factor')
        self.add_input('Area', 10.0, units='m**2', desc='Rotor disc area')
        self.add_input('rho', 1.225, units='kg/m**3', desc='Air density')
        self.add_input('Vu', 10.0, units='m/s', desc='Freestream air velocity, upstream of rotor')

        # Outputs
        self.add_output('Vr', 0.0, units='m/s', desc='Air velocity at rotor exit plane')
        self.add_output('Vd', 0.0, units='m/s', desc='Slipstream air velocity, downstream of rotor')
        self.add_output('Ct', 0.0, desc='Thrust coefficient')
        self.add_output('Cp', 0.0, desc='Power coefficient')
        self.add_output('power', 0.0, units='W', desc='Power produced by the rotor')
        self.add_output('thrust', 0.0, units='m/s')

        self.declare_partials('Vr', ['a', 'Vu'])
        self.declare_partials('Vd', 'a')
        self.declare_partials('Ct', 'a')
        self.declare_partials('thrust', ['a', 'Area', 'rho', 'Vu'])
        self.declare_partials('Cp', 'a')
        self.declare_partials('power', ['a', 'Area', 'rho', 'Vu'])
```

```
def compute(self, inputs, outputs):
    a = inputs['a']
    Vu = inputs['Vu']
    rho = inputs['rho']
    Area = inputs['Area']
    qA = 0.5 * rho * Area * Vu ** 2
    outputs['Vd'] = Vd = Vu * (1 - 2 * a)
    outputs['Vr'] = 0.5 * (Vu + Vd)
    outputs['Ct'] = Ct = 4 * a * (1 - a)
    outputs['thrust'] = Ct * qA
    outputs['Cp'] = Cp = Ct * (1 - a)
    outputs['power'] = Cp * qA * Vu

def compute_partials(self, inputs, J):
    a = inputs['a']
    Vu = inputs['Vu']
    Area = inputs['Area']
    rho = inputs['rho']

    a_times_area = a * Area
    one_minus_a = 1.0 - a
    a_area_rho_vu = a_times_area * rho * Vu

    J['Vr', 'a'] = -Vu
    J['Vr', 'Vu'] = one_minus_a
    J['Vd', 'a'] = -2.0 * Vu
    J['Ct', 'a'] = 4.0 - 8.0 * a
    J['thrust', 'a'] = 0.5 * rho * Vu**2 * Area * J['Ct', 'a']
    J['thrust', 'Area'] = 2.0 * Vu**2 * a * rho * one_minus_a
    J['thrust', 'Vu'] = 4.0 * a_area_rho_vu * one_minus_a
    J['Cp', 'a'] = 4.0 * a * (2.0 * a - 2.0) + 4.0 * one_minus_a**2
    J['power', 'a'] = 2.0 * Area * Vu**3 * a * rho * (2.0 * a - 2.0) + 2.0 * Area * Vu**3 * rho * one_minus_a**2
    J['power', 'Area'] = 2.0 * Vu**3 * a * rho * one_minus_a ** 2
    J['power', 'rho'] = 2.0 * a_times_area * Vu ** 3 * (one_minus_a)**2
    J['power', 'Vu'] = 6.0 * Area * Vu**2 * a * rho * one_minus_a**2
```

In OpenMDAO, multiple components can be connected together inside of a Group. There will be some other new elements to review, so let's take a look:

Betz Group:

In [3]:
```
class Betz(om.Group):
    """
    Group containing the actuator disc equations for deriving the Betz limit.
    """

    def setup(self):
        indeps = self.add_subsystem('indeps', om.IndepVarComp(), promotes=['*'])
        indeps.add_output('a', 0.5)
        indeps.add_output('Area', 10.0, units='m**2')
        indeps.add_output('rho', 1.225, units='kg/m**3')
        indeps.add_output('Vu', 10.0, units='m/s')

        self.add_subsystem('a_disk', ActuatorDisc(), promotes=['a', 'Area', 'rho', 'Vu'])
```

The `Betz` class derives off of the OpenMDAO `Group` class, which is typically the top-level class that is used in an analysis. The OpenMDAO `Group` class allows you to cluster models in hierarchies. We can put multiple components in groups. We can also put other groups in groups.

Components are added to groups with the `self.add_subsystem` command, which has two primary arguments. The first is the string name to call the subsystem that is added and the second is the component or sub-group class instance. A common optional argument is `promotes=`, which elevatest the input/ouput variable string names to the top-level namespace. The `Betz` group shows examples where the `promotes=` can be passed a list of variable string names or the `'*'` wildcard to mean all input/output variables.

The first subsystem that is added is an `IndepVarComp`, which are the independent variables of the problem. Subsystem inputs that are not tied to other subsystem outputs should be connected to an independent variables. For optimization problems, design variables must be part of an `IndepVarComp`. In the Betz problem, we have `a`, `Area`, `rho`, and `Vu`. Note that they are promoted to the top level namespace, otherwise we would have to access them by `'indeps.x'` and `'indeps.z'`.

Let's optimize our system!

Even though we have all the pieces in a `Group`, we still need to put them into a `Problem` to be executed. The `Problem` instance is where we can assign design variables, objective functions, and constraints. It is also how the user interacts with the `Group` to set initial conditions and interrogate output values.

First, we instantiate the `Problem` and assign an instance of `Betz` to be the root model:

[4]:
```
prob = om.Problem()
prob.model = Betz()
```

Next we assign an optimization driver to the problem instance. If we only wanted to evaluate the model once and not optimize, then a driver is not needed:

[5]:
```
prob.driver = om.ScipyOptimizeDriver()
prob.driver.options['optimizer'] = 'SLSQP'
```

With the optimization driver in place, we can assign design variables, objective(s), and constraints. Any IndepVarComp can be a design variable and any model output can be an objective or constraint.

We want to maximize the objective, but OpenMDAO will want to minimize it as it is consistent with the standard optimization problem statement. So we minimize the negative to find the maximum. Note that `Cp` is not promoted from `a_disk`. Therefore we must reference it with `a_disk.Cp`

[6]:
```
prob.model.add_design_var('a', lower=0.0, upper=1.0)
prob.model.add_design_var('Area', lower=0.0, upper=1.0)
prob.model.add_objective('a_disk.Cp', scaler=-1.0)
```

Let's optimize our system!

Even though we have all the pieces in a `Group`, we still need to put them into a `Problem` to be executed. The `Problem` instance is where we can assign design variables, objective functions, and constraints. It is also how the user interacts with the `Group` to set initial conditions and interrogate output values.

First, we instantiate the `Problem` and assign an instance of `Betz` to be the root model:

[4]:
```
prob = om.Problem()
prob.model = Betz()
```

Next we assign an optimization driver to the problem instance. If we only wanted to evaluate the model once and not optimize, then a driver is not needed:

[5]:
```
prob.driver = om.ScipyOptimizeDriver()
prob.driver.options['optimizer'] = 'SLSQP'
```

With the optimization driver in place, we can assign design variables, objective(s), and constraints. Any IndepVarComp can be a design variable and any model output can be an objective or constraint.

We want to maximize the objective, but OpenMDAO will want to minimize it as it is consistent with the standard optimization problem statement. So we minimize the negative to find the maximum. Note that `Cp` is not promoted from `a_disk`. Therefore we must reference it with `a_disk.Cp`

[6]:
```
prob.model.add_design_var('a', lower=0.0, upper=1.0)
prob.model.add_design_var('Area', lower=0.0, upper=1.0)
prob.model.add_objective('a_disk.Cp', scaler=-1.0)
```

Now we can run the optimization:

In [7]:
```
prob.setup()
prob.run_driver()
```

```
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -0.5925925906659251
            Iterations: 5
            Function evaluations: 6
            Gradient evaluations: 5
Optimization Complete
-----------------------------------
```

Out[7]:
```
False
```

Finally, the result:

Above, we see a summary of the steps in our optimization. Don't worry about the output `False` for now. Next, we print out the values we care about and list all of the inputs and outputs that are problem used.

In [8]:
```
print('Coefficient of power Cp = ', prob['a_disk.Cp'])
print('Induction factor a =', prob['a'])
print('Rotor disc Area =', prob['Area'], 'm^2')
all_inputs  = prob.model.list_inputs(values=True)
all_outputs = prob.model.list_outputs(values=True)
```

```
Coefficient of power Cp =  [0.59259259]
Induction factor a = [0.33335528]
Rotor disc Area = [1.] m^2
```

Tutorial 4: Modeling a whole turbine and plant

- (Brief) Introduction to WISDEM and OpenMDAO
- WISDEM installation (essentials only)
- Tutorial 1: Run a simple WISDEM calculation with Jupyter Notebooks
- Some more detail on OpenMDAO
- Tutorial 2: Finding the Betz Limit through OpenMDAO optimization
- **Tutorial 4: Modeling a whole turbine and plant**

jupyter

Quit | Logout

Files | Running | Clusters

Select items to perform actions on them.

Upload | New |

0 /	Name	Last Modified	File size
	img	5 days ago	
	01_cost_and_scaling.ipynb	5 days ago	63.4 kB
	02_betz_limit.ipynb	5 days ago	18.1 kB
	03_sellar.ipynb	5 days ago	18.7 kB
	04_turbine_assembly.ipynb	5 days ago	35.2 kB
	WISDEM_notebooks.pdf	5 days ago	804 kB

Tutorial 4: Turbine Assembly

Here's what we've done so far in these tutotirals:

- Ran two simple cost models of turbines. In these, we estiamted masses of components and cost per kilogram of those components.
- We learned how OpenMDAO makes *components* when we calculated the Betz limit by modelling an idealized `ActuatorDisc` as a subclass of `ExplicitComponent`.
- We learned how to group multiple components into groups with the OpenMDAO `Group` class when we modelled the Sellar problem.

We can now turn our attention back to WISDEM and put together a rotor, drivetrain and tower to model a complete wind turbine. We will use the tools we have gained so far in these tutorials to accomplish this.

This is a significant increase in complexity from our previous toy examples. This tutorial doesn't aim to give an exhaustive line-by-line explanation of nearly 400 lines of source code. However, these fundamental building blocks of components, groups and susbsytems are used to model systems of significant complexity.

First, we need to import our dependencies

There are many dependencies we need to import. Of key interesst to use here are various parts of WISDEM that we will assemble into our model.

```
from wisdem.rotorse.rotor import RotorSE, Init_RotorSE_wRefBlade
from wisdem.rotorse.rotor_geometry_yaml import ReferenceBlade
from wisdem.towerse.tower import TowerSE
from wisdem.commonse import NFREQ
from wisdem.commonse.environment import PowerWind, LogWind
from wisdem.commonse.turbine_constraints import TurbineConstraints
from wisdem.turbine_costsse.turbine_costsse_2015 import Turbine_CostsSE_2015
from wisdem.plant_financese.plant_finance import PlantFinance
from wisdem.drivetrainse.drivese_omdao import DriveSE
```

Thank you

www.nrel.gov

NREL/PR-5000-75659

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 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.

NREL
NATIONAL RENEWABLE ENERGY LABORATORY