WindSE Documentation

Release 2022.10.0_dev

Ryan King, Jeffery Allen, Ethan Young

Oct 09, 2022

Contents:

CHAPTER 1

Installation

It is easies to run WindSE within a conda environment. To install conda check this link: Conda Installation. Additionally, WindSE has been tested on MacOS Catalina (10.15), but in theory should also run on linux. Windows is not recommended.

1.1 Source Conda Installation (Script):

The easiest way to install windse is to run:

```
sh install.sh <enviroment_name>
```

Then the enviroment can be activated using:

```
conda activate <enviroment_name>
```

1.2 Source Conda Installation (Manual):

If you want to use the latest version or just want to setup the environment manually, follow these steps. After conda is installed, create a new environment using:

```
conda create --name <enviroment_name>
```

You can replace the name <enviroment_name> with a different name for the environment if you want. Next we activate the environment using:

```
conda activate <enviroment_name>
```

or whatever you named your environment. Now we need to install the dependent packages using:

```
conda install -c conda-forge fenics=2019.1.0=py38_9 dolfin-adjoint matplotlib scipy=1.
↪4.1 slepc mshr hdf5 pyyaml memory_profiler pytest pytest-cov pytest-mpi coveralls
```

Next, we need to install the tsfc form compilers::

```
pip install git+https://github.com/blechta/tsfc.git@2018.1.0
pip install git+https://github.com/blechta/COFFEE.git@2018.1.0
pip install git+https://github.com/blechta/FInAT.git@2018.1.0
pip install git+https://github.com/mdolab/pyoptsparse@v1.0
pip install singledispatch networkx pulp openmdao
```

Finally, download/clone the WindSE repo and run:

```
pip install -e .
```

in the root folder.

CHAPTER 2

Running WindSE

To run WindSE, first create a parameters file (as described in *The Parameter File* and demonstrated in *Demos*). Then activate the conda environment using:

```
source activate <enviroment_name>
```

where the <enviroment_name> is what was defined in the install process then run:

```
windse run <params file>
```

where <params files> is the path of the parameters file you wish to run. By default windse searches for "params.txt" in the current directory if no file is supplied.

Sit back and let the magic happen. Additionally, you can run:

```
windse run <params file> -p group:option:value
```

where `group:option:value` is a single unbroken string and the group is the group in the params file, the option is the specific option in that group and the value is the new value. This allows for overriding parameters in the yaml file via the terminal. For example: `wind_farm:HH:140` will change the hub height of all turbines in a "grid" or "random" farm to 140 m regardless of what was defined in the params.yaml file.

CHAPTER 3

The Parameter File

This is a comprehensive list of all the available parameters. The default values are stored within `default_parameters.yaml` located in the windse directory of the python source files.

3.1 Adding a New Parameter

To add a new parameter, first add an entry in the `default_parameters.yaml` file under one of the major sections with a unique name. The value of that entry will then be an attribute of the class associated with that section. For example: adding the entry `test_param: 42` under the `wind_farm` section will add the attribute `self.test_param` to the `GenericWindFarm` class with the default value of 42.

3.2 General Options

This section is for options about the run itself. The basic format is:

```
general:
    name:               <str>
    preappend_datetime: <bool>
    output:             <str list>
    output_folder:      <str>
    output_type:        <str>
    dolfin_adjoint:     <bool>
    debug_mode:         <bool>
```

Option	Description	Required	Default
`name`	Name of the run and the folder in `output/`.	no	"Test"
`preappend_datetime`	Append the date to the output folder name.	no	False
`output`	Determines which functions to save. Select any combination of the following: "mesh", "initial_guess", "height", "turbine_force", "solution", "debug"	no	["solution"]
`output_folder`	The folder location for all the output	no	"output/"
`output_type`	Output format: "pvd" or "xdmf".	no	"pvd"
`dolfin_adjoint`	Required if performing any optimization.	no	False
`debug_mode`	Saves "tagged_output.yaml" full of debug data	no	False

3.3 Domain Options

This section will define all the parameters for the domain:

```
domain:
    type:               <str>
    path:               <str>
    mesh_path:          <str>
    terrain_path:       <str>
    bound_path:         <str>
    filetype:           <str>
```

(continues on next page)

(continued from previous page)

```
    scaled:              <bool>
    ground_reference:    <float>
    streamwise_periodic: <bool>
    spanwise_periodic:   <bool>
    x_range:             <float list>
    y_range:             <float list>
    z_range:             <float list>
    nx:                  <int>
    ny:                  <int>
    nz:                  <int>
    mesh_type:           <str>
    center:              <float list>
    radius:              <float>
    nt:                  <int>
    res:                 <int>
    interpolated:        <bool>
    analytic:            <bool>
    gaussian:
        center:   <float list>
        theta:    <float>
        amp:      <float>
        sigma_x:  <float>
        sigma_y:  <float>
    plane:
        intercept: <float list>
        mx:        <float>
        my:        <float>
```

Option	Description	Required (for)	Default	Units
`type`	Sets the shape/dimension of the mesh. Choices: "rectangle", "box", "cylinder", "circle" "imported", "interpolated"	yes	None	-
`path`	Folder of the mesh data to import	yes or `*_path` "imported"	`*_path`	-
`mesh_path`	Location of specific mesh file Default file name: "mesh"	no "imported"	`path`	-
`terrain_path`	Location of specific terrain file Default file name: "terrain.txt" Note: Only file required by "interpolated"	no "imported"	`path`	-
`bound_path`	Location of specific boundary marker data Default file name: "boundaries"	no "imported"	`path`	-
`filetype`	file type for imported mesh: "xml.gz", "h5"	no "imported"	"xml.gz"	-
`scaled`	Scales the domain to km instead of m. WARNING: extremely experimental!	no	False	-
`ground_reference`	The height (z	no	0.0	m

gaussian	If analytic is true, a Gaussian hill will be created using the following parameters. Note: requires interpolated and analytic.	"interpolated" "analytic"	None	-
center	The center point of the gaussian hill.	no	[0.0,0.0]	m
amp	The amplitude of the hill.	yes	None	m
sigma_x	The extent of the hill in the x direction.	yes	None	m
sigma_y	The extent of the hill in the y direction.	yes	None	m
theta	The rotation of the hill.	no	0.0	rad

plane	If analytic is true, the ground will be represented as a plane Note: requires interpolated and analytic.	"interpolated" "analytic"	None	-
intercept	The equation of a plane intercept	no	[0.0,0.0,0.0]	m
mx	The slope in the x direction	yes	None	m
my	The slope in the y direction	yes	None	m

To import a domain, three files are required:

- mesh.xml.gz - this contains the mesh in a format dolfin can handle
- boundaries.xml.gz - this contains the facet markers that define where the boundaries are
- topology.txt - this contains the data for the ground topology.

The topology file assumes that the coordinates are from a uniform mesh. It contains three column: x, y, z. The x and y columns contain just the unique values. The z column contains the ground values for every combination of x and y. The first row must be the number of points in the x and y direction. Here is an example for z=x+y/10:

```
3 3 9
0 0 0.0
1 1 0.1
```

(continues on next page)

(continued from previous page)

```
2 2 0.2
    1.0
    1.1
    1.2
    2.0
    2.1
    2.2
```

Note: If using "h5" file format, the mesh and boundary will be in one file.

3.4 Wind Farm Options

This section will define all the parameters for the wind farm:

```
wind_farm:
    type:               <str>
    path:               <str>
    display:            <str>
    ex_x:               <float list>
    ex_y:               <float list>
    x_spacing:          <float>
    y_spacing:          <float>
    x_shear:            <float>
    y_shear:            <float>
    min_sep_dist:       <float>
    grid_rows:          <int>
    grid_cols:          <int>
    jitter:             <float>
    numturbs:           <int>
    seed:               <int>
```

Option	Description	Required (for)	Default	Units
`type`	Sets the type of farm. Choices: “grid”, “random”, “imported”, “empty”	yes	None	-
`path`	Location of the wind farm csv file	“imported”	None	-
`display`	Displays a plot of the wind farm	no	False	-
`ex_x`	The x extents of the farm where turbines can be placed	“grid” “random”	None	m
`ex_y`	The y extents of the farm where turbines can be placed	“grid” “random”	None	m
`x_spacing`	Alternative method for defining grid farm x distance between turbines	“grid”	None	m
`y_spacing`	Alternative method for defining grid farm y distance between turbines	“grid”	None	m
`x_shear`	Alternative method for defining grid farm offset in the x direction between rows	no “grid”	None	m
`y_shear`	Alternative method for defining grid farm offset in the y direction between columns	no “grid”	None	m

To import a wind farm, set the path to a .csv file containing the per turbine information. In the .csv file, each column specifies a turbine property and each row is a unique turbine. At minimum, the locations for each turbine must be specified. Here is a small two turbine example:

```
     x,      y
200.00, 0.0000
800.00, 0.0000
```

Additional turbine properties can be set by adding a column with a header equal to the yaml parameter found in the "Turbine Options" section. Here is an example of a two turbine farm with additional properties set:

```
  x,       y,      HH,      yaw,      RD, thickness,   axial
0.0,  -325.0,   110.0,   0.5236,   130.0,      13.0,    0.33
0.0,   325.0,   110.0,  -0.5236,   130.0,      13.0,    0.33
```

The columns can be in any order and white space is ignored. If a property is set in both the yaml and the imported .csv, the value in the .csv will be used and a warning will be displayed.

3.5 Turbine Options

This section will define all the parameters for the wind farm:

```
turbines:
    type:               <str>
    HH:                 <float>
    RD:                 <float>
    thickness:          <float>
    yaw:                <float>
    axial:              <float>
    force:              <str>
    rpm:                <float>
    read_turb_data:     <str>
    blade_segments:     <int or str>
    use_local_velocity: <bool>
    max_chord:          <float>
    chord_factor:       <float>
    gauss_factor:       <float>
```

Option	Description	Required (for)	Default	Units
type	Sets the type of farm. Choices: "disk" - actuator disk representation using the FEniCS backend "2D_disk" - actuator disk representation optimized for 2D simulations "numpy_disk" - actuator disk representation that uses numpy arrays "line" - actuator line representation best used with the unsteady solver	yes	None	-
HH	The hub height of the turbine from ground	all	None	m
RD	The rotor diameter	all	None	m
yaw	Determines the yaw of all turbines. Yaw is relative to the wind inflow direction	all	None	rad
thickness	The effective thickness of the rotor disk	"disk" or disk variant	None	m
axial	The axial induction factor	"disk" or disk variant	None	-
force	the radial distribution of force Choices: "sine", "constant"	no "disk"	"sine"	-
rpm	sets the revolutions per minute if using	"line"	10.0	rev/min

See "Wind Farm Options" for how to specify turbine properties individually for each turbine.

3.6 Refinement Options

This section describes the options for refinement The domain created with the previous options can be refined in special ways to maximize the efficiency of the number DOFs. None of these options are required. There are three types of mesh manipulation: warp, farm refine, turbine refine. Warp shifts more cell towards the ground, refining the farm refines within the farm extents, and refining the turbines refines within the rotor diameter of a turbine. When choosing to warp, a "smooth" warp will shift the cells smoothly towards the ground based on the strength. A "split" warp will attempt to create two regions, a high density region near the ground and a low density region near the top

The options are:

```
refine:
    warp_type:          <str>
    warp_strength:      <float>
    warp_percent:       <float>
    warp_height:        <float>
    farm_num:           <int>
    farm_type:          <str>
    farm_factor:        <float>
    turbine_num:        <int>
    turbine_type:       <str>
    turbine_factor:     <float>
    refine_custom:      <list list>
    refine_power_calc: <bool>
```

Option	Description
warp_type	Choose to warp the mesh to place more cells near the ground. Choices: "smooth", "split"
warp_strength	The higher the strength the more cells moved towards the ground. Requires: "smooth"
warp_percent	The percent of the cell moved below the warp height. Requires: "split"
warp_height	The height the cell are moved below Requires: "split"
farm_num	Number of farm refinements
farm_type	The shape of the refinement around the farm Choices: "full" - refines the full mesh "box" - refines in a box near the farm "cylinder" - cylinder centered at the farm "stream" - stream-wise cylinder around farm (use for 1 row farms)
farm_factor	A scaling factor to make the refinement area larger or smaller
turbine_num	Number of turbine refinements
turbine_type	The shape of the refinement around turbines Choices: "simple" - cylinder around turbine "tear" - tear drop shape around turbine "wake" - cylinder to capture wake
turbine_factor	A scaling factor to make the refinement area larger or smaller
refine_custom	This is a way to define multiple refinements in a specific order allowing for more complex refinement options. Example below
refine_power_calc	bare minimum refinement around turbines to

To use the "refine_custom" option, define a list of lists where each element defines refinement based on a list of parameters. Example:

```
refine_custom: [
    [ "full",     [ ]                                  ],
    [ "full",     [ ]                                  ],
    [ "box",      [ [[-500,500],[-500,500],[0,150]] ]  ],
    [ "cylinder", [ [0,0,0], 750, 150 ]                ],
    [ "simple",   [ 100 ]                              ],
    [ "tear",     [ 50, 0.7853 ]                       ]
]
```

For each refinement, the first option indicates how many time this specific refinement will happen. The second option indicates the type of refinement: "full", "square", "circle", "farm_circle", "custom". The last option indicates the extent of the refinement.

The example up above will result in five refinements:

1. Two full refinements
2. One box refinement bounded by: [[-500,500],[-500,500],[0,150]]
3. One cylinder centered at origin with radius 750 m and a height of 150 m
4. One simple turbine refinement with radius 100 m
5. One teardrop shaped turbine refinement radius 500 m and rotated by 0.7853 rad

The syntax for each refinement type is:

```
[ "full",     [ ]                                                                ]
[ "box",      [ [[x_min,x_max],[y_min,y_max],[z_min,z_max]], expand_factor ]  ]
[ "cylinder", [ [c_x,c_y,c_z], radius, height, expand_factor ]                   ]
[ "stream",   [ [c_x,c_y,c_z], radius, length, theta, offset, expand_factor ] ]
[ "simple",   [ radius, expand_factor ]                                          ]
[ "tear",     [ radius, theta, expand_factor ]                                   ]
[ "wake",     [ radius, length, theta, expand_factor ]                           ]
```

Note:

- For cylinder, the center is the base of the cylinder
- For stream, the center is the start of the vertical base and offset indicates the rotation offset
- For stream, wake, length is the distance center to the downstream end of the cylinder
- For stream, tear, wake, theta rotates the shape around the center

3.7 Function Space Options

This section list the function space options:

```
function_space:
    type: <str>
    quadrature_degree: <int>
    turbine_space:     <str>
    turbine_degree:    <int>
```

Option	Description	Required	Default
`type`	Sets the type of farm. Choices: "linear": P1 elements for both velocity and pressure "taylor_hood": P2 for velocity, P1 for pressure	yes	None
`quadrature_degree`	Sets the quadrature degree for all integration and interpolation for the whole simulation	no	6
`turbine_space`	Sets the function space for the turbine. Only needed if using "numpy" for `turbine_method` Choices: "Quadrature", "CG"	no	Quadrature
`turbine_degree`	The quadrature degree for specifically the turbine force representation. Only works "numpy" method Note: if using Quadrature space, this value must equal the `quadrature_degree`	no	6

3.8 Boundary Condition Options

This section describes the boundary condition options. There are three types of boundary conditions: inflow, no slip, no stress. By default, inflow is prescribed on boundary facing into the wind, no slip on the ground and no stress on all other faces. These options describe the inflow boundary velocity profile.

```
boundary_conditions:
    vel_profile:    <str>
    HH_vel:         <float>
    vel_height:     <float, str>
    power:          <float>
    k:              <float>
    turbsim_path    <str>
    inflow_angle:   <float, list>
    boundary_names:
        east:       <int>
        north:      <int>
        west:       <int>
        south:      <int>
        bottom:     <int>
        top:        <int>
        inflow:     <int>
        outflow:    <int>
    boundary_types:
        inflow:     <str list>
        no_slip:    <str list>
        free_slip:  <str list>
        no_stress:  <str list>
```

Option	Description	Required	Default
`vel_profile`	Sets the velocity profile. Choices: "uniform": constant velocity of u_{HH} "power": a power profile "log": log layer profile "turbsim": use a turbsim simulation as inflow	yes	None
`HH_vel`	The velocity at hub height, u_{HH}, in m/s.	no	8.0
`vel_height`	sets the location of the reference velocity. Use "HH" for hub height	no	"HH"
`power`	The power used in the power flow law	no	0.25
`k`	The constant used in the log layer flow	no	0.4
`inflow_angle`	Sets the initial inflow angle for the boundary condition. A multiangle solve can be indicated by setting this value to a list with values: [start, stop, n] where the solver will perform n solves, sweeping uniformly through the start and stop angles. The number of solves, n, can also be defined in the solver parameters.	no	None
`turbsim_path`	The location of turbsim profiles used as inflow boundary conditions	yes "turbsim"	None
`boundary_names`	A dictionary used to identify the boundaries	no	See Below
`boundary_types`	A dictionary for defining boundary conditions	no	See Below

If you are importing a mesh or want more control over boundary conditions, you can specify the boundary markers using `names` and `types`. The default for these two are

Rectangular Mesh:

```
boundary_condition:
    boundary_names:
        east:  1
        north: 2
        west:  3
        south: 4
    boundary_types:
        inflow:    ["west","north","south"]
        no_stress: ["east"]
```

Box Mesh:

```
boundary_condition:
    boundary_names:
        east:   1
        north:  2
        west:   3
        south:  4
        bottom: 5
        top:    6
    boundary_types:
        inflow:    ["west","north","south"]
        free_slip: ["top"]
        no_slip:   ["bottom"]
        no_stress: ["east"]
```

Circle Mesh:

```
boundary_condition:
    boundary_names:
        outflow: 7
        inflow:  8
    boundary_types:
        inflow:    ["inflow"]
        no_stress: ["outflow"]
```

Cylinder Mesh:

```
boundary_condition:
    boundary_names:
        outflow: 5
        inflow:  6
        bottom:  7
        top:     8
    boundary_types:
        inflow:    ["inflow"]
        free_slip: ["top"]
        no_slip:   ["bottom"]
        no_stress: ["outflow"]
```

These defaults correspond to an inflow wind direction from West to East.

When marking a rectangular/box domains, from a top-down perspective, start from the boundary in the positive x direction and go counter clockwise, the boundary names are: “easy”, “north”, “west”, “south”. Additionally, in 3D

there are also "top" and "bottom". For a circular/cylinder domains, the boundary names are "inflow" and "outflow". Likewise, in 3D there are also "top" and "bottom". Additionally, you can change the `boundary_types` if using one of the built in domain types. This way you can customize the boundary conditions without importing a whole new mesh.

3.9 Problem Options

This section describes the problem options:

```
problem:
    type:                 <str>
    use_25d_model:        <bool>
    viscosity:            <float>
    lmax:                 <float>
    turbulence_model:     <str>
    script_iterator:      <int>
    use_corrective_force: <bool>
    stability_eps:        <float>
```

Option	Description	Required	Default
`type`	Sets the variational form use. Choices: "taylor_hood": Standard RANS formulation "stabilized": Adds a term to stabilize P1xP1 formulations	yes	None
`viscosity`	Kinematic Viscosity	no	0.1
`lmax`	Turbulence length scale	no	15.0
`use_25d_model`	Option to enable a small amount of compressibility to mimic the effect of a 3D, out-of-plane flow solution in a 2D model.	no "2D only"	False
`turbulence_model`	Sets the turbulence model. Choices: mixing_length, smagorinsky, or None	no	mixing_length
`script_iterator`	debugging tool, do not use	no	0
`use_corrective_force`	add a force to the weak form to allow the inflow to recover	no	False
`stability_eps`	stability term to help increase the well-posedness of the linear mixed formulation	no	1.0

3.10 Solver Options

This section lists the solver options:

```
solver:
    type:               <str>
    pseudo_steady:      <bool>
    final_time:         <float>
    save_interval:      <float>
    num_wind_angles:    <int>
    endpoint:           <bool>
    velocity_path:      <str>
    power_type:         <str>
    save_power:         <bool>
    nonlinear_solver:   <str>
    newton_relaxation: <float>
    cfl_target: 0.5     <float>
    cl_iterator: 0      <int>
```

Option	Description	Required (for)	Default
`type`	Sets the solver type. Choices: "steady": solves for the steady state solution "iterative_steady": uses iterative SIMPLE solver "unsteady": solves for a time varying solution "multiangle": iterates through inflow angles uses `inflow_angle` or [0, 2π] "imported_inflow": runs multiple steady solves with imported list of inflow conditions	yes	None
`pseudo_steady`	used with unsteady solver to create a iterative steady solver.	no "unsteady"	False
`final_time`	The final time for an unsteady simulation	no "unsteady"	1.0 s
`save_interval`	The amount of time between saving output fields	no "unsteady"	1.0 s
`num_wind_angles`	Sets the number of angles. can also be set in `inflow_angle`	no "multiangle"	1
`endpoint`	Should the final inflow angle be simulated	no "multiangle"	False
`velocity_path`	The location of a list of inflow conditions	yes "imported_inflow"	
`power_type`	Sets the power functional Choices: "power": simple	no	"power"

The "multiangle" solver uses the steady solver to solve the RANS formulation. Currently, the "multiangle" solver does not support imported domains.

3.11 Optimization Options

This section lists the optimization options. If you are planning on doing optimization make sure to set `dolfin_adjoint` to True.

```
optimization:
    opt_type:          <str>
    control_types:     <str list>
    layout_bounds:     <float list>
    objective_type:    <str, str list, dict>
    save_objective:    <bool>
    opt_turb_id :      <int, int list, str>
    record_time:       <str, float>
    u_avg_time:        <float>
    opt_routine:       <string>
    obj_ref:           <float>
    obj_ref0:          <float>
    taylor_test:       <bool>
    optimize:          <bool>
    gradient:          <bool>
    constraint_types: <dict>
```

Option	Description	Required	Default
`opt_type`	Type of optimization: "minimize" or "maximize"	no	maximize
`control_types`	Sets the parameters to optimize. Choose Any: "yaw", "axial", "layout", "lift", "drag", "chord"	yes	None
`layout_bounds`	The bounding box for the layout optimization	no	wind_farm
`objective_type`	Sets the objective function for optimization. Visit `windse.objective_functions()` to see choices and additional keywords. See below to an example for how to evaluate multiple objectives. The first objective listed will always be used in the optimization.	no	power
`save_objective`	Save the value of the objective function output/`name`/data/objective_data.txt Note: power objects are saved as power_data.txt	no	True
`opt_turb_id`	Sets which turbines to optimize Choices: int: optimize single turbine by ID list: optimize all in list by ID "all": optimize all	no	all
`record_time`	The amount of time to run the simulation before calculation of the objective function takes place Choices: "computed": let the	no unsteady	computed

The `objective_type` can be defined in three ways. First as a single string such as:

```
optimization:
    objective_type: alm_power
```

If the object chosen in this way has any keyword arguments, the defaults will automatically chosen. The second way is as a list of strings like:

```
optimization:
    objective_type: ["alm_power", "KE_entrainment", "wake_center"]
```

Again, the default keyword argument will be used with this method. The final way is as a full dictionary, which allow for setting keyword arguments:

```
optimization:
    objective_type:
        power: {}
        point_blockage:
            location: [0.0,0.0,240.0]
        plane_blockage_#1:
            axis: 2
            thickness: 130
            center: 240.0
        plane_blockage_#2:
            axis: 0
            thickness: 130
            center: -320.0
        cyld_kernel:
            type: above
        mean_point_blockage:
            z_value: 240
```

Notice that since the objective named "power" does not have keyword arguments, an empty dictionary must be passed. For a full list of objective function visit: `windse.objective_functions()`. Notice that we can have multiple version of the same objective by appending the name with "_#" and then a number. This allows us to evaluate objectives of the same type with different keyword arguments. Regardless of the number of objective types listed, currently, only the first one will be used for an optimization.

The `constraint_types` option is defined in a similar way. By default the minimum distance between turbines is setup:

```
constraint_types:
    min_dist:
        target: 2
        scale:  1
```

This constraint will only be used if the `control_types` contains "layout". Additional constraints can be added using the same objective functions from `windse.objective_functions()` by setting:

```
constraint_types:
    min_dist:
        target: 2
        scale:  1
    plane_blockage:
        target: 8.0
        scale: -1
        kwargs:
```

(continues on next page)

(continued from previous page)

```
            axis: 2
            thickness: 130
            center: 240.0
```

This will still enforce the layout constraint but will additionally enforce a "plane_blockage" type constraint. By default, the constrains are setup like:

$$s * (c(m) - t) \geq 0$$

where c is the constraint function, t is the target, s is the scale, and m are the controls. In this configuration, we are enforcing that the result of the constraint function is greater than or equal to the target. However, we can set the scale to -1 to flip the inequality. Just like the `objective_type`, multiple constraints of the same type can be use by appending "_#" followed by a number to the end of the name with the exception of the "min_dist" type.

CHAPTER 4

Demos

4.1 Example Parameter Files

These examples show how to use the parameters file. See *The Parameter File* page for more details. All of these examples can be run using `windse run <file>`. Some file require inputs, which can be downloaded `here`.

1. `2D Simulations`
2. `2D Layout Optimization`
3. `3D Simulations`
4. `Multi-Angle Simulations`
5. `Yaw Optimization`
6. `Multi-Angle Optimization`
7. `Actuator Line Method Single-Turbine Simulation`

Note: These demos are extremely coarse to lower runtime for automated testing. To get better results, increase the mesh resolution and try different refinements.

4.2 Example Driver Files

These examples show how you build a custom driver if desired. Check the *WindSE API* for details on the available functions.

1. Constructing a Gridded Wind Farm on a 2D rectangular domain: *2D Demo*.

4.3 Related Pages

4.3.1 Gridded Wind Farm on a Rectangular Domain

This demonstration will show how to set up a 2D rectangular mesh with a wind farm consisting of a 36 turbines laid out in a 6x6 grid. This demo is associated with two files:

- Parameter File: `params.yaml`
- Driver File: `2D_Grid_driver.py`

Setting up the parameters:

To write a WindSE driver script, we first need to define the parameters. This must be completed before building any WindSE objects. There are two way to define the parameters:

1. Loading a parameters yaml file
2. Manually creating the parameter dictionary directly in the driver.

Both methods will be discussed below and demonstrated in the next section.

The parameter file:

First we will discuss the parameters file method. The parameter file is the main way to customize a simulation. The driver file uses the options specified in the parameters file to run the simulation. Ideally, multiple simulations can use a single driver file and multiple parameter files.

The parameter file is formated as a yaml structure and requires pyyaml to be read. The driver file is written in python.

The parameter file is broken up into several sections: general, domain, boundaries, and wind_farm, etc.

The full parameter file can be found here: `params.yaml` and more information can be found here: *Parameter File Explained*.

Manual parameter dictionary:

The manual method involve creating a blank nested dictionary and populating it with the parameters needed for the simulation. The *`windse_driver.driver_functions.BlankParameters()`* will create the blank nested dictionary for you.

Creating the driver code:

The full driver file can be found here: `2D_Grid_driver.py` First, we start off with the import statements:

```
import windse
import windse_driver.driver_functions as df
```

Next, we need to set up the parameters. If we want to load them from a yaml file we would run:

```
# windse.initialize("params.yaml")
# params = windse.windse_parameters
```

However, in this demo, we will define the parameters manually. Start by creating a blank parameters object:

```
params = df.BlankParameters()
```

Next, populate the general options:

```
params["general"]["name"]        = "2D_driver"
params["general"]["output"]      = ["mesh","initial_guess","turbine_force","solution"]
params["general"]["output_type"] = "xdmf"
```

Then, the wind farm options:

```
params["wind_farm"]["type"]      = "grid"
params["wind_farm"]["grid_rows"] = 6
params["wind_farm"]["grid_cols"] = 6
params["wind_farm"]["ex_x"]      = [-1800,1800]
params["wind_farm"]["ex_y"]      = [-1800,1800]
params["wind_farm"]["HH"]        = 90
params["wind_farm"]["RD"]        = 126
params["wind_farm"]["thickness"] = 10
params["wind_farm"]["yaw"]       = 0
params["wind_farm"]["axial"]     = 0.33
```

and the domain options:

```
params["domain"]["type"]    = "rectangle"
params["domain"]["x_range"] = [-2500, 2500]
params["domain"]["y_range"] = [-2500, 2500]
params["domain"]["nx"]      = 50
params["domain"]["ny"]      = 50
```

Lastly, we just need to define the type of boundary conditons, function space, problem formulation and solver we want:

```
params["boundary_conditions"]["vel_profile"] = "uniform"
params["function_space"]["type"] = "taylor_hood"
params["problem"]["type"]        = "taylor_hood"
params["solver"]["type"]         = "steady"
```

Now that the dictionary is set up, we need to initialize WindSE:

```
params = df.Initialize(params)
```

That was basically the hard part. Now with just a few more commands, our simulation will be running. First we need to build the domain and wind farm objects:

```
dom, farm = df.BuildDomain(params)
```

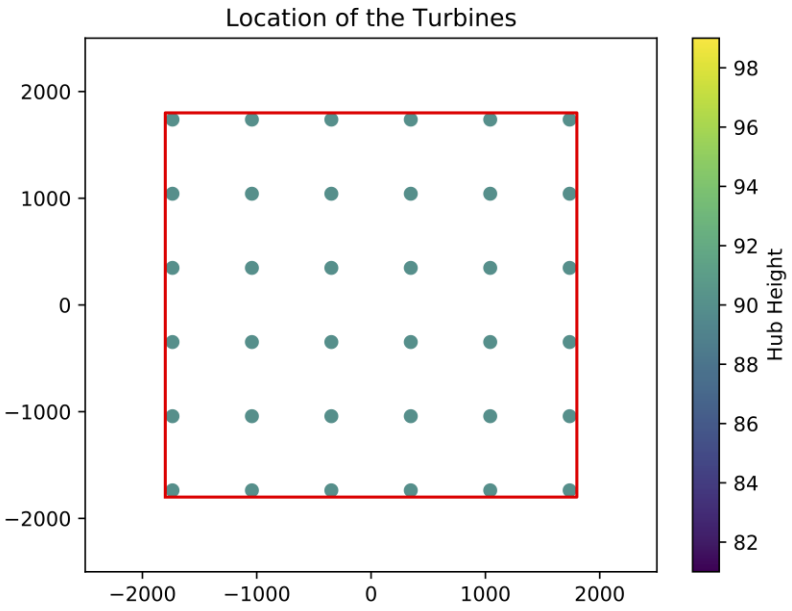
We can inspect the wind farm by running:

```
farm.Plot(True)
```

This results in a wind farm that looks like this:

Alternatively, we could have use `False` to generate and save the plot, but not display it. This is useful for running batch test or on a HPC. We could also manually save the mesh using `dom.Save()`, but since we specified the mesh as an output in the parameters file, this will be done automatically when we solve.

Next, we need to setup the simulation problem:




```
problem = df.BuildProblem(params,dom,farm)
```

For this problem we are going to use Taylor-Hood elements, which are comprised of 2nd order Lagrange elements for velocity and 1st order elements for pressure.

The last step is to build the solver:

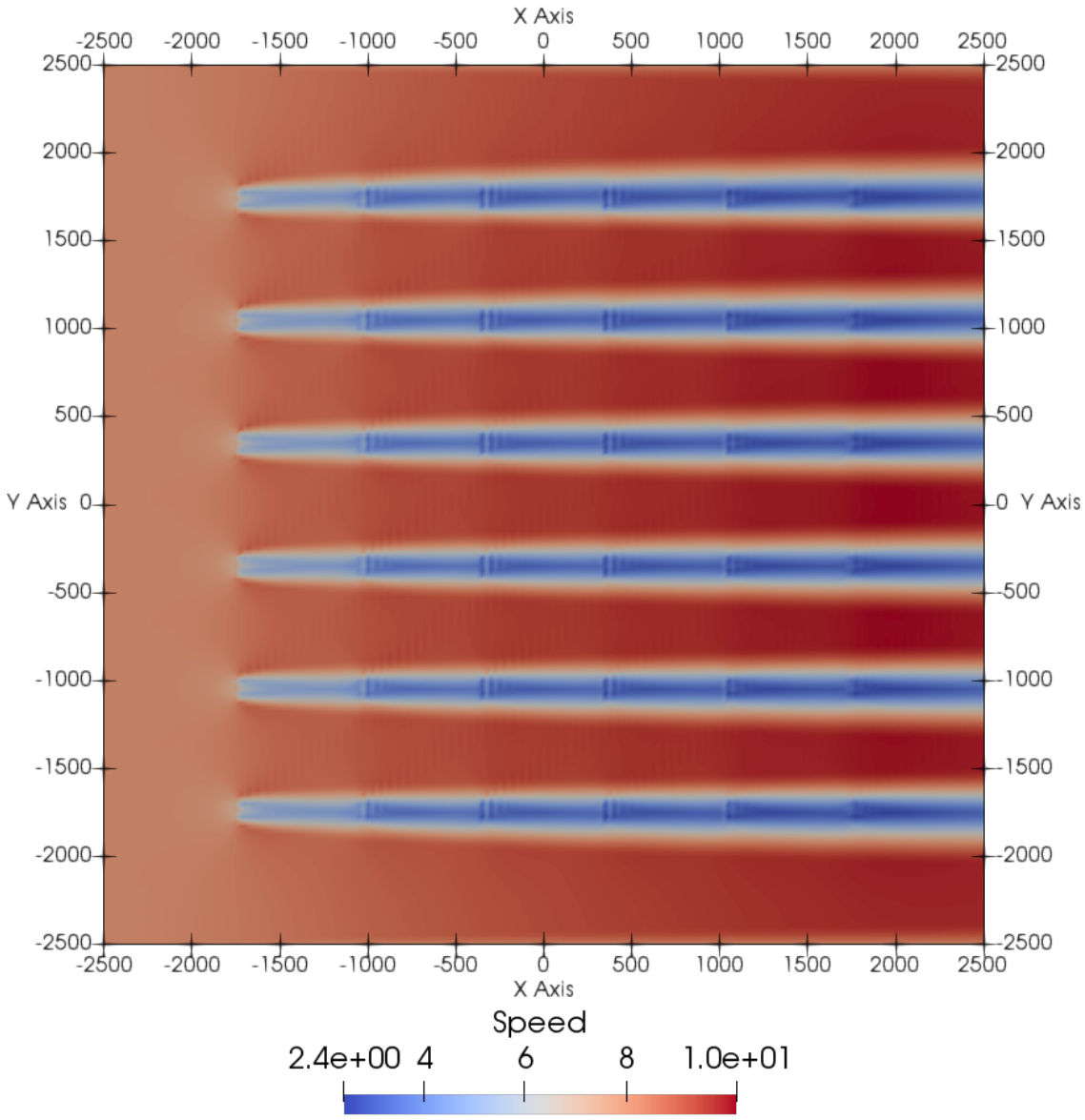
```
solver = df.BuildSolver(params,problem)
```

This problem has uniform inflow from the west. The east boundary is our outflow and has a no-stress boundary condition.

Finally, it's time to solve:

```
solver.Solve()
```

Running `solver.Solve()` will save all the inputs according to the parameters file, solve the problem, and save the solution. If everything went smoothly, the solution for wind speed should be:



4.3.2 Setting up general options:

The general options are those that will effect the entire run and usually specify how to handle i/o. for this demo the general parameters are:

```
general:
  name: "2D"
  preappend_datetime: false
  output: ["mesh","initial_guess","turbine_force","solution"]
  output_type: "xdmf"
```

The `name` parameter determines the naming structure for the output folders. usually the output folder is `output/`

<name>/. This is the only required options.

Setting preappend_datetime to true will append the name with a datetime stamp. This is useful when running multiple simulation as they will be organized by date. The default option for this is false

The outputs is a list of function that will be saved when solver.Solve() is called. These strings can be in any combination:

- mesh: saves the mesh and boundary markers
- initial_guess: saves the initial velocity and pressure used by the Newton iteration
- height: saves a function indicating the terrain height and depth
- turbine_force: saves the function that is used to represent the turbines
- solution: saves the velocity and pressure after a solve

By default, the only output is solution.

Finally, the output_type is the file format for the saved function. Currently WindSE supports xdmf and pvd with the latter being the default. However, the mesh files are always saved in the pvd format.

4.3.3 Setting up the domain:

Next we need to set the parameters for the domain:

```
domain:
  #                    # Description           | Units
  x_range: [-2500, 2500] # x-range of the domain | m
  y_range: [-2500, 2500] # y-range of the domain | m
  nx: 200                # Number of x-nodes     | -
  ny: 200                # Number of y-nodes     | -
```

This will create a mesh that has 200 nodes in the x-direction and 200 nodes in the y-direction. The mesh will be a rectangle with side lengths of 5000 m and centered at (0,0).

4.3.4 Setting up the wind farm:

The last step for this demo is to set up the wind farm:

```
wind_farm:
  #                  # Description              | Units
  ex_x: [-1800,1800] # x-extent of the farm     | m
  ex_y: [-1800,1800] # y-extent of the farm     | m
  grid_rows: 6       # Number of rows           | -
  grid_cols: 6       # Number of columns        | -
  yaw: 0             # Yaw                      | rads
  axial: 0.33        # Axial Induction          | -
  HH: 90             # Hub Height               | m
  RD: 126            # Turbine Diameter         | m
  thickness: 10      # Effective Thickness      | m
```

This will produce a 6 by 6 grid evenly spaced in an area of [-1800,1800] X [-1800,1800]. Note that ex_x X ex_y is the extent of the farm and should be a subset of the domain ranges. The extent accounts for the rotor diameter to ensure all turbines including the rotors are located within the extents. The rest of the parameters determine the physical properties of the turbines:

- yaw: The yaw of the turbines where 0 is perpendicular to an East to West inflow.

- `axial`: The axial induction
- `HH`: The hub height relative to the ground
- `RD`: The rotor diameter
- `thickness`: The effective thickness of the rotor used for calculating the turbine force

4.3.5 Other Required Parameters:

Additionally, we need to specify a few parameters that are required for some checks. These options are not actually used within the custom driver:

```
problem:
  type: taylor-hood

solver:
  type: steady
```

CHAPTER 5

Studies

This is a list of studies performed with WindSE.

5.1 Actuator Disk Mesh Convergence

This study is designed to develop intuition on how refined the actuator disk model needs to be to produce converged power.

5.1.1 Keywords:

mesh, actuator disk, power

5.1.2 Input files and code version:

This study was ran using this parameter file and code version:

- Parameter File: `../../../demo/documented/studies/disk_mesh_convergence/simulation/power.yaml`
- Code Version: WindSE 2021.08.01

5.1.3 Setup:

This simulation starts with a 3x3 grid arrangement of turbines with 3 rotor diameters padding for the inflow, outflow and sides as seen in Figure 1. The initial mesh has 16x16x10 cells in the x, y, and z directions, respectively. The mesh is then refined up to 3 times to get the mesh seen in Figure 2. Each refinement is local in a cylinder centered on each turbine with a radius of 1.25 time the rotor diameter and extending the full height of the turbine. For each level of refinement, a steady RANS simulation is performed with log layer inflow with hub height inflow speed of 8 m/s with the wind blowing from west to east. The number of refinement was controlled using the command line override parameter:

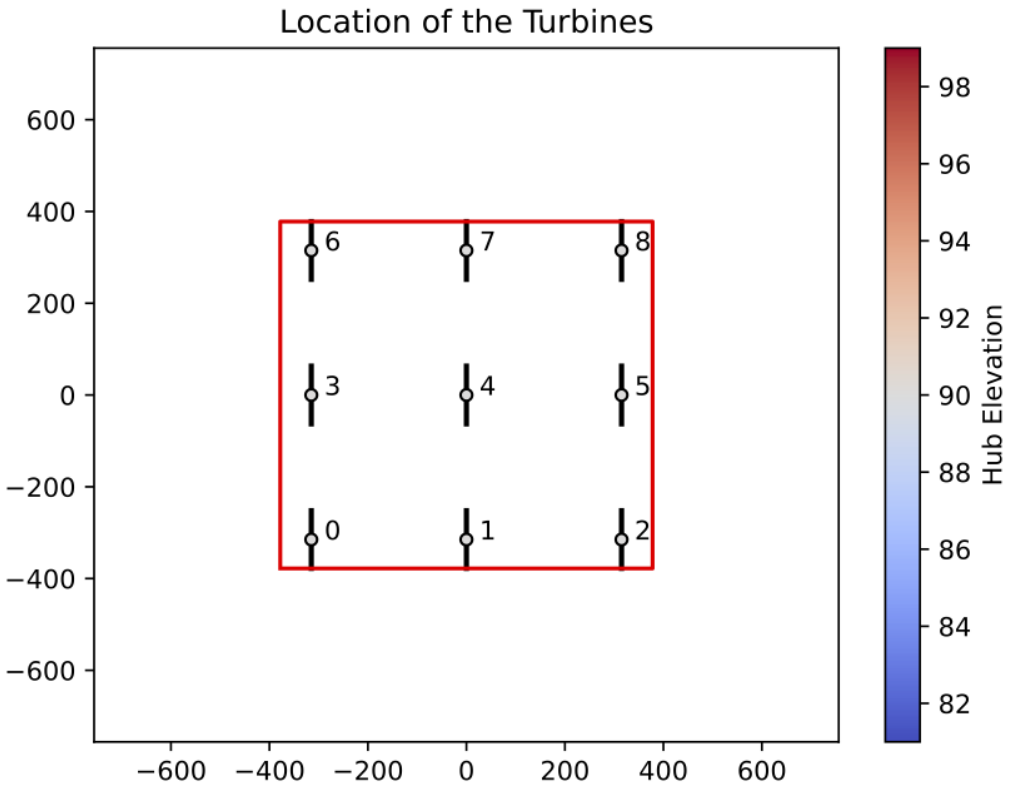


Fig. 1: Figure 1: The wind farm layout

```
windse run power.yaml -p general:name:n=N -p refine:turbine_num:N
```

where N is the number of refinements.

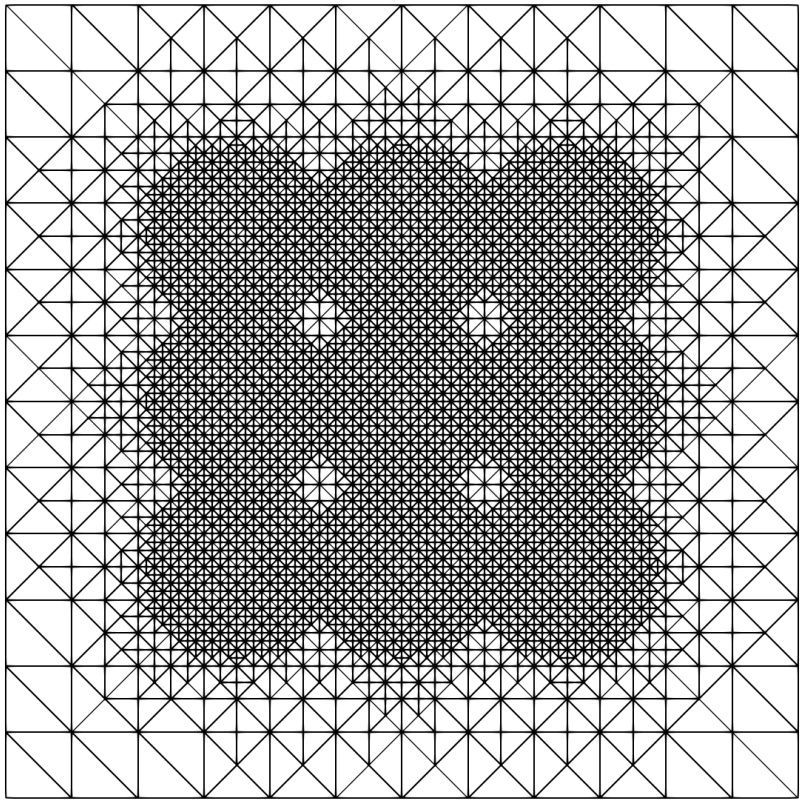


Fig. 2: Figure 2: The mesh after 3 turbine refinements

5.1.4 Results:

The full compiled power output for each refinement level can be found in Table 1. The "mesh spacing" column is calculated by taking the full width of the mesh (1512 m) and dividing it by the initial number of cells in the x direction (16), which results in 94.5 m. This a measurement of the distance between mesh nodes. If we divide the rotor diameter by the mesh spacing, we get an approximation for the number of mesh nodes that span an actuator disk. This number is useful for determining how well resolved the disks are with a given resolution. For example, after 3 turbine refinements a disk is represented by about 10 nodes in the mesh. The goal of this study is to determine how many nodes per turbine is necessary to produced converged power calculations.

Table 1: Table 1: Power data for each turbine and refinement level

Refinements	DOFs (Thousands)	Mesh Spacing (m)	Nodes/Turbine	Turbine_0	Turbine_1	Turbine_2	Turbine_3	Turbine_4	Turbine_5	Turbine_6	Turbine_7	Turbine_8	Total
0	12.72	94.50	1.33	2.31	0.55	0.43	2.27	0.52	0.44	2.46	0.53	0.41	9.92
1	49.10	47.25	2.67	1.66	0.49	0.38	1.82	0.46	0.37	1.65	0.48	0.38	7.69
2	285.50	23.63	5.33	1.72	0.50	0.33	1.72	0.50	0.33	1.72	0.50	0.33	7.64
3	1820.62	11.81	10.67	1.73	0.51	0.31	1.73	0.52	0.31	1.73	0.51	0.31	7.68

Note: The magnitude of the power produced is not part of this study and has not been calibrated. This study is exclusively looking at mesh convergence.

First let's look a the total power produced in Figure 3. Looking at this farm scale metric implies that convergence is essentially reached after one level of refinement. Looking at the "nodes/turbine" column in Table 1, this corresponds to needing only ~3 nodes per turbine, which seems exceptionally low.

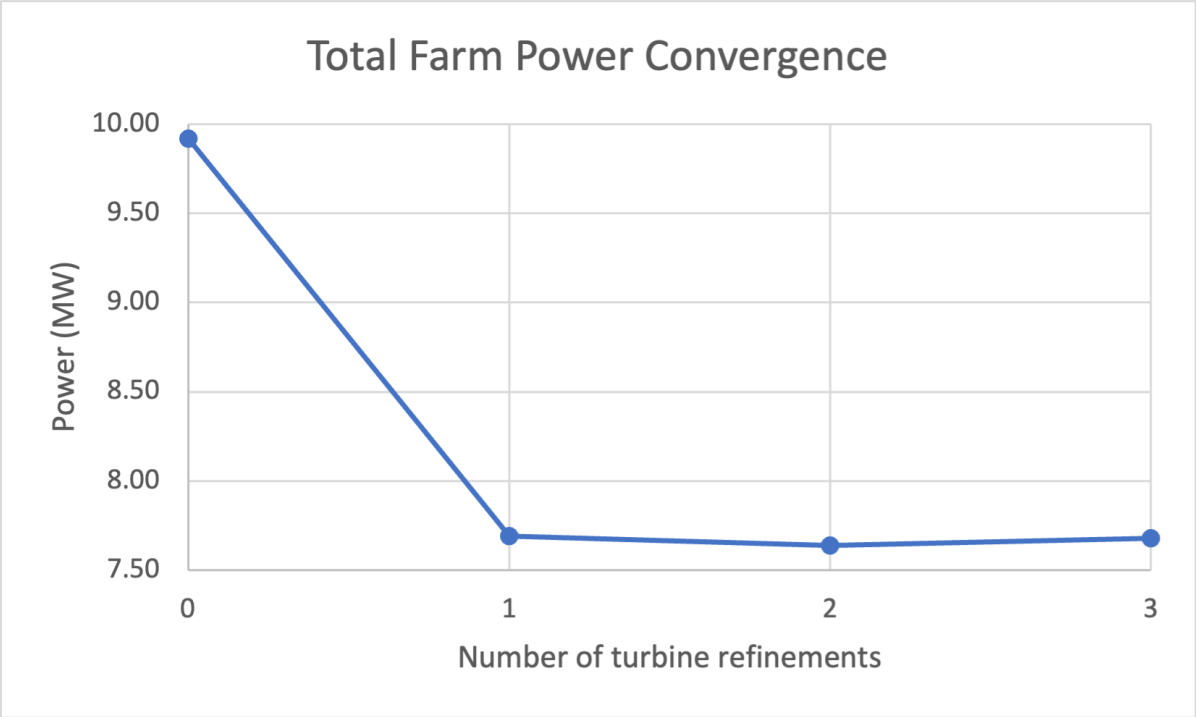


Fig. 3: Figure 3: Total power with respect to mesh resolution

The story doesn't end there though. We can also look at the power produced by the leading turbines and fully waked turbines. Because the wind is blowing west to east the leading turbine are numbers 0, 3, 6 and the turbines we are calling "fully waked" are numbers 2, 5, 8. The leading turbine power shown in Figure 4 shows that convergence is a bit slower than the full farm's power taking an additional refinement. It is also interesting to note that all three turbines converge to the same power. This is expected because the inflow profile is not turbulent and uniform in the y direction so each of the leading turbines should experience the exact same forces resulting in identical powers.

Finally, looking at the fully waked power production in Figure 5, we see a completely different trend. Now it is possible that these turbines are not yet fully converged. It appears that the power is converging but might require an additional refinement for a total of 4. Currently all of these simulation are running on a laptop, which does not have enough memory to run the 4 refinement simulation. This implies that if the wakes are exceptionally important to the simulation, more refinement is required. That said, after only 3 refinement, all three fully waked turbines produce the same power, just like the leading turbines. Since this is also expected, this could indicate 3 refinements or about 10 nodes per turbine is sufficient.

5.1.5 Conclusions:

Based on the information presented in this study, we conclude that when performing a steady simulation with actuator disk, aim for around 10 mesh nodes per turbine to get the best computational performance to accuracy. Some future

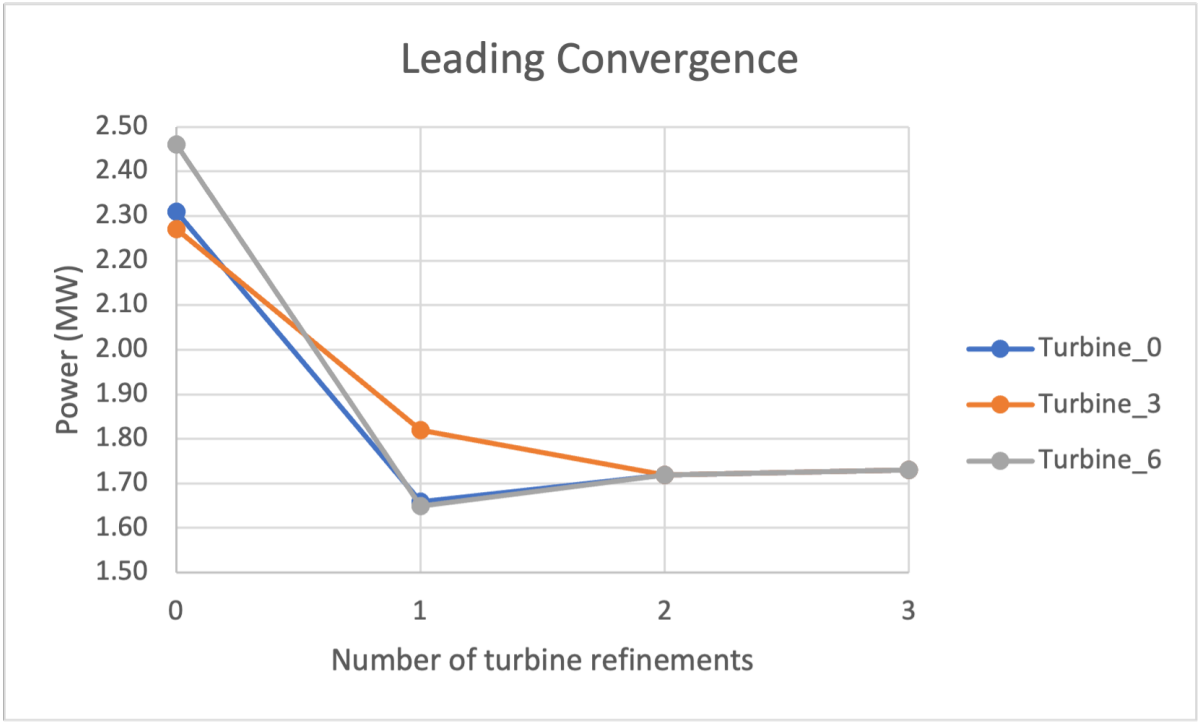


Fig. 4: Figure 4: Power of the leading edge of turbines

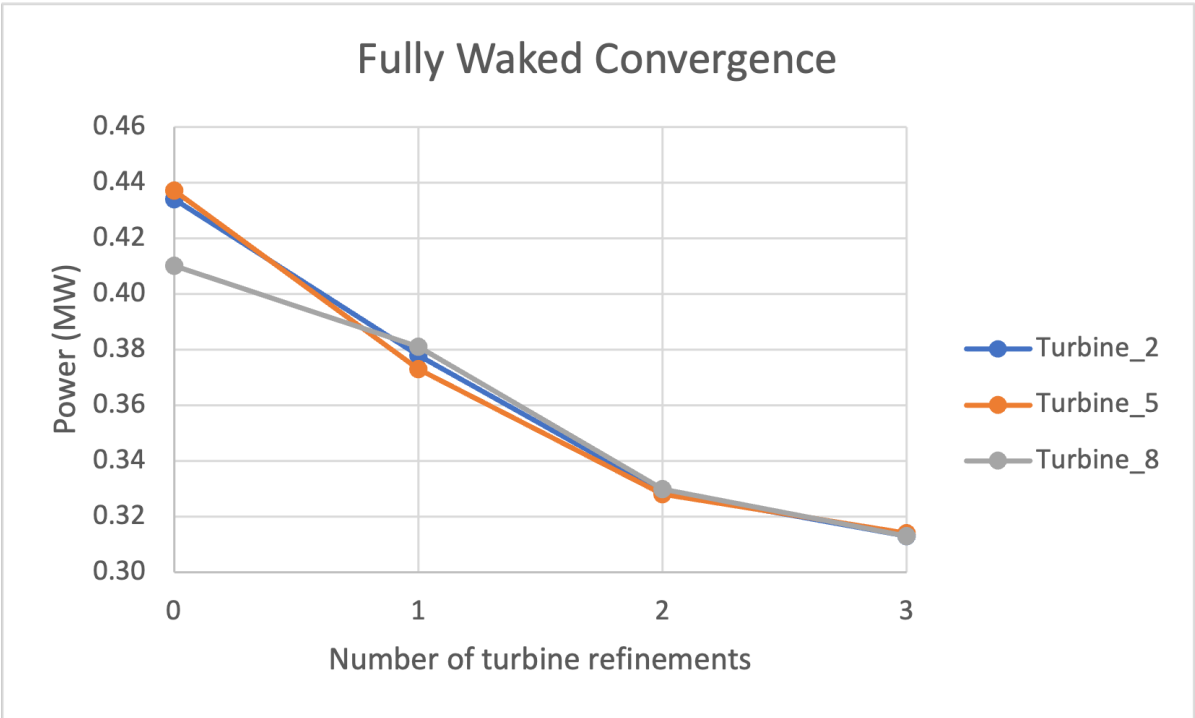


Fig. 5: Figure 5: Power of the fully wake turbines

studies that would be useful to refine this recommendation would include investigating mesh convergence of power with respect to:

- turbulent inflow
- increasing number of waked turbines
- refining waked turbines more than leading turbines

5.2 Actuator Line Method Validation

In this study, we compare four key along-blade forces calculated by WindSE's actuator line method—namely lift, drag, angle of attack, and axial velocity—to a benchmark 2018 study by Martínez-Tossas et al.[1]

5.2.1 Keywords:

actuator line method, ALM, lift, drag, angle of attack, velocity, validation

5.2.2 Input files and code version:

The complete details of the domain, mesh, and turbine can be found in Martínez-Tossas et al.[1] but are reproduced as a WindSE study in the following YAML file.

- Parameter File: `../../../demo/documented/studies/alm_validation/input_files/alm_validation.yaml`
- Code Version: WindSE 2021.08.01

5.2.3 Setup:

The setup used for this test and enumerated in the YAML file reproduces almost exactly the setup of Martínez-Tossas et al.[1] The turbine used is the NREL 5-MW reference turbine which has a rotor diameter, D, of 126 m operating with rotational speed 9.155 RPM. The computational domain spans $-3D \leq x \leq 21D$, $-3D \leq y \leq 3D$, $-3D \leq z \leq 3D$ where the rotor is centered at $(0, 0, 0)$ and oriented normal to the x^+ flow direction. The grid size surrounding the rotor is 1.96875 m, which implies 64 mesh cells across the rotor diameter. 64 actuator nodes are used along the length of each blade with a fixed Gaussian size of 10 m. The fluid is assigned a density $\rho = 1.0$ kg/m^3, the inflow velocity is a uniform 8 m/s applied at the x^- wall, slip boundary conditions (no flow through) are set at all lateral walls, and a 0-pressure outlet condition applied at the x^+ wall. The Smagorinsky eddy viscosity model is used with $C_s = 0.16$.

Note that because we are only comparing along-blade quantities in this study and not features in the far wake, we make a slight deviation from the paper and perform *local* mesh refinements to acheive the specified 1.96875 m grid size in the region surrounding the turbine. This reproduces the mesh size and resolution around the rotor as specified in the paper but leaves relatively coarse mesh cells in the downstream region to reduce the computational workload. By this same token, we only run the simulations long enough for the along-blade quantities to converge, rather than the much longer time needed to satisfy repeated flow throughs, which was experimentally found to be 100 s.

[1] Luis A. Martínez-Tossas, Matthew J. Churchfield, Ali Emre Yilmaz, Hamid Sarlak, Perry L. Johnson, Jens N. Sørensen, Johan Meyers, and Charles Meneveau, "Comparison of four large-eddy simulation research codes and effects of model coefficient and inflow turbulence in actuator-line-based wind turbine modeling", Journal of Renewable and Sustainable Energy 10, 033301 (2018) https://doi.org/10.1063/1.5004710

5.2.4 Results:

The results generated by the WindSE ALM implementation are plotted alongside 4 other codes, including NREL's SOWFA, in Figure 1:

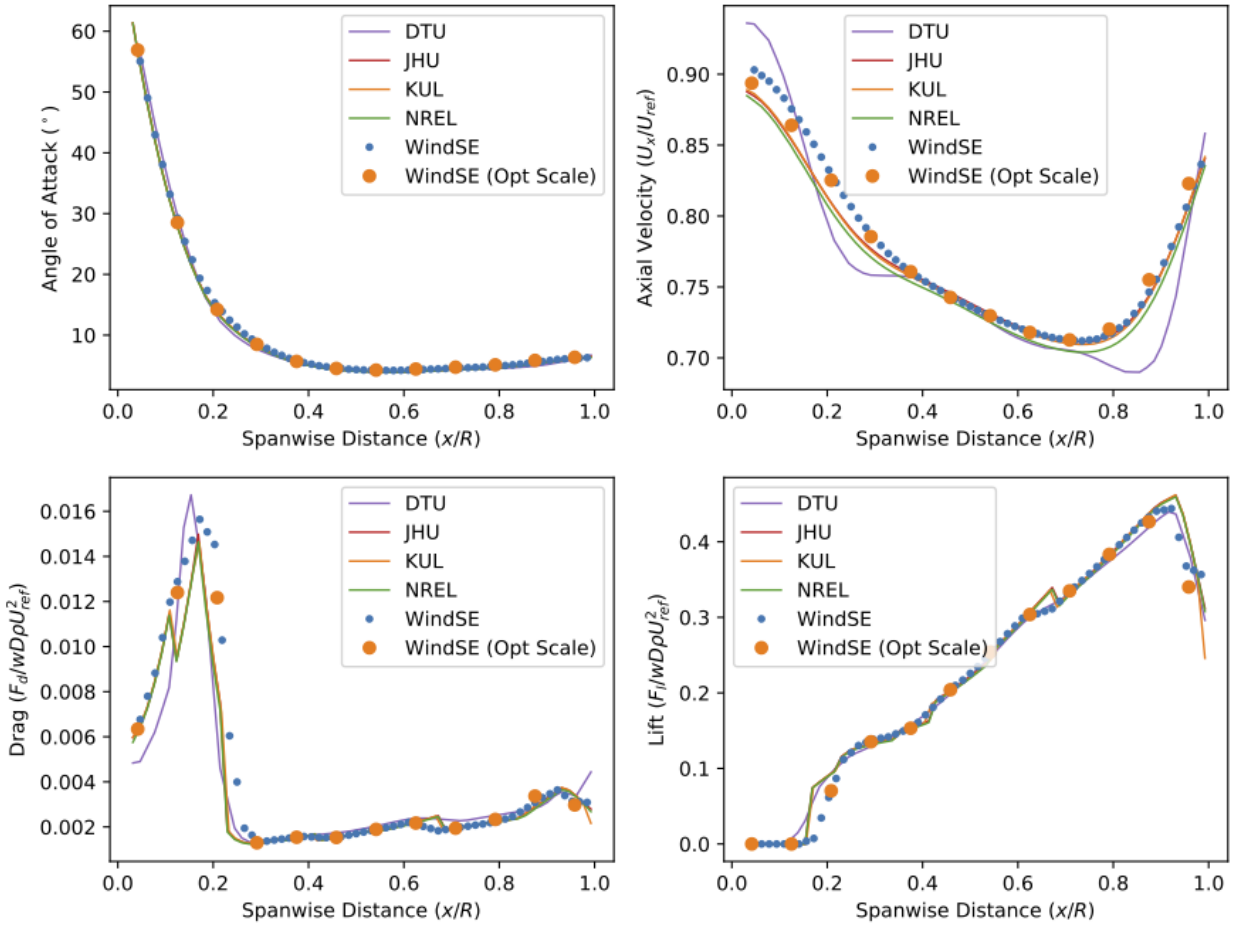


Fig. 6: Figure 1: Clockwise from top left, the angle of attack, relative axial velocity, lift, and drag as a function of position along the blade calculated using two different discretizations: the 64-actuator version outlined above and a 12-acuator version testing a greatly reduced number of control points that would be more suitable for an optimization problem (opt scale) featuring actuator-based controls.

Both the 64-node and 12-node, optimization scale, cases recover the along-blade forces well. The axial velocity is normalized by the reference velocity, $U_{ref} = 8$ m/s, and so represents the percentage of freestream while the lift and drag are non-dimensionalized by the actuator length, w, rotor diameter, density, and reference velocity.

5.2.5 Conclusions:

We find that along-blade forces are recovered very well using WindSE's actuator line implementation. The largest deviations away from the comparison codes seem to occur in regions of sharp change (see the first 20% of blade span drag profile) where differences in the airfoil section resolution and the interpolation used between airfoil sections may have a large effect.

With respect to sizing ALM simulations, we are able to produce very good agreement with both the benchmark 64-node setup and the 12-node, optimization scale, setup. Although it is important to keep in mind that we are unlikely

to produce such good agreement in the downstream wake using this study, for the purposes of optimizing objectives confined to the rotor plane (e.g., a single turbine's power) with respect to along-the-blade quantities like chord or twist angle, using ~10 actuator nodes sized to be roughly ~1/10 of the rotor diameter seems to be a good starting point. For a more detailed convergence study of actuator node resolution and size, see the 2017 paper of Martínez-Tossas et al.[2]

5.2.6 References:

[2] Martínez-Tossas, L. A., Churchfield, M. J., and Meneveau, C. (2017) Optimal smoothing length scale for actuator line models of wind turbine blades based on Gaussian body force distribution. Wind Energ., 20: 1083– 1096. doi: 10.1002/we.2081.

CHAPTER 6

WindSE API

6.1 windse_driver.driver_functions

windse_driver.driver_functions.**BlankParameters**()
returns a nested dictionary that matches the first level of the parameters dictionary

windse_driver.driver_functions.**BuildDomain**(*params*)
This function build the domain and wind farm objects.

Parameters **params** (windse.Parameters) – an overloaded dict containing all parameters.

Returns

- **dom** (windse.GenericDomain) – the domain object that contains all mesh related information.
- **farm** (windse.GenericWindFarm) – the wind farm object that contains the turbine information.

windse_driver.driver_functions.**BuildProblem**(*params*, *dom*, *farm*)
This function compiles everything into a single problem object and build the variational problem functional.

Parameters

- **params** (*windse.Parameters*) – an overloaded dict containing all parameters.
- **dom** (*windse.GenericDomain*) – the domain object that contains all mesh related information.
- **farm** (*windse.GenericWindFarm*) – the wind farm object that contains the turbine information.

Returns contains all information about the simulation.

Return type problem (windse.GenericProblem)

windse_driver.driver_functions.**BuildSolver**(*params*, *problem*)
This function builds the solver object. Solve with solver.Solve()

Parameters

- **params** (*windse.Parameters*) – an overloaded dict containing all parameters.
- **problem** (*windse.GenericProblem*) – contains all information about the simulation.

Returns **solver** – contains the solver routines.

Return type windse.GenericSolver

windse_driver.driver_functions.**DefaultParameters**()
return the default parameters list

windse_driver.driver_functions.**Initialize**(*params_loc=None*)
This function initialized the windse parameters.

Parameters **params_loc** (*str*) – the location of the parameter yaml file.

Returns **params** – an overloaded dict containing all parameters.

Return type windse.Parameters

windse_driver.driver_functions.**SetupSimulation**(*params_loc=None*)
This function automatically sets up the entire simulation. Solve with solver.Solve()

Parameters **params_loc** (*str*) – the location of the parameter yaml file.

Returns

- **params** (*windse.Parameters*) – an overloaded dict containing all parameters.
- **problem** (*windse.GenericProblem*) – contains all information about the simulation.
- **solver** (*windse.GenericSolver*) – contains the solver routines. Solve with solver.Solve()

6.1.1 Functions

windse_driver.driver_functions.**BlankParameters**()
returns a nested dictionary that matches the first level of the parameters dictionary

windse_driver.driver_functions.**BuildDomain**(*params*)
This function build the domain and wind farm objects.

Parameters **params** (windse.Parameters) – an overloaded dict containing all parameters.

Returns

- **dom** (windse.GenericDomain) – the domain object that contains all mesh related information.

- **farm** (windse.GenericWindFarm) – the wind farm object that contains the turbine information.

windse_driver.driver_functions.**BuildProblem**(*params*, *dom*, *farm*)

This function compiles everything into a single problem object and build the variational problem functional.

Parameters

- **params** (*windse.Parameters*) – an overloaded dict containing all parameters.
- **dom** (*windse.GenericDomain*) – the domain object that contains all mesh related information.
- **farm** (*windse.GenericWindFarm*) – the wind farm object that contains the turbine information.

Returns contains all information about the simulation.

Return type problem (windse.GenericProblem)

windse_driver.driver_functions.**BuildSolver**(*params*, *problem*)

This function builds the solver object. Solve with solver.Solve()

Parameters

- **params** (*windse.Parameters*) – an overloaded dict containing all parameters.
- **problem** (*windse.GenericProblem*) – contains all information about the simulation.

Returns **solver** – contains the solver routines.

Return type windse.GenericSolver

windse_driver.driver_functions.**DefaultParameters**()

return the default parameters list

windse_driver.driver_functions.**Initialize**(*params_loc=None*)

This function initialized the windse parameters.

Parameters **params_loc** (*str*) – the location of the parameter yaml file.

Returns **params** – an overloaded dict containing all parameters.

Return type windse.Parameters

windse_driver.driver_functions.**SetupSimulation**(*params_loc=None*)

This function automatically sets up the entire simulation. Solve with solver.Solve()

Parameters **params_loc** (*str*) – the location of the parameter yaml file.

Returns

- **params** (*windse.Parameters*) – an overloaded dict containing all parameters.
- **problem** (*windse.GenericProblem*) – contains all information about the simulation.
- **solver** (*windse.GenericSolver*) – contains the solver routines. Solve with solver.Solve()

CHAPTER 7

Indices and tables

- genindex
- modindex

Python Module Index

w

Index