WindSE Documentation

Release 2021.08.01

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", "interpo-lated"	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 im-ported 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>
    HH:                     <float>
    RD:                     <float>
    thickness:              <float>
    yaw:                    <float>
    axial:                  <float>
    force:                  <str>
    turbine_method:         <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: "grid", "random", "imported"	yes	None	-
`path`	Location of the wind farm text 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, create a .txt file with this formatting:

```
#     x       y      HH     Yaw    Diameter Thickness Axial_Induction
200.00 0.0000 80.000  0.000      126.0      10.5            0.33
800.00 0.0000 80.000  0.000      126.0      10.5            0.33
```

The first row isn't necessary. Each row defines a different turbine.

3.5 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

3.6 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

3.7 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.

3.8 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

3.9 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.

3.10 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>
```

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:
            axis: 2
            thickness: 130
            center: 240.0
        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()*

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)
```

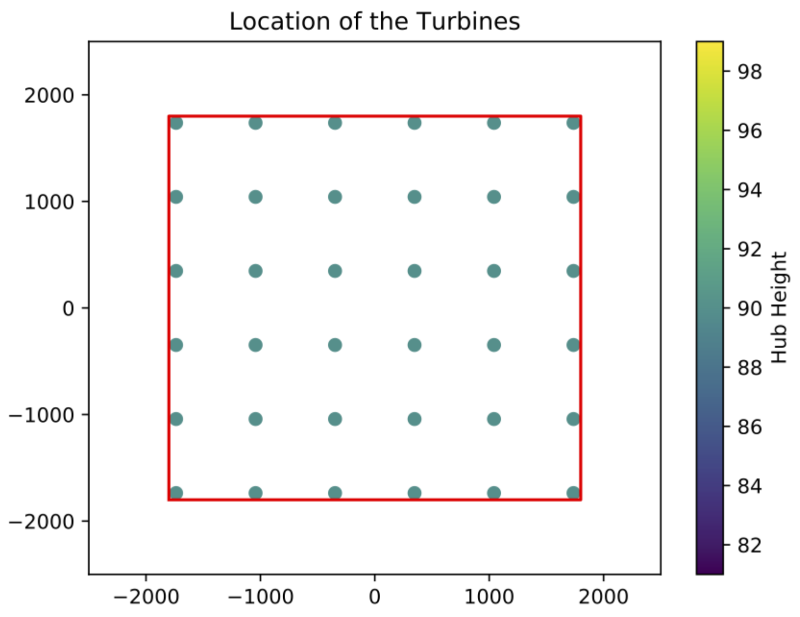
We can inspect the wind farm by running:

```
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:

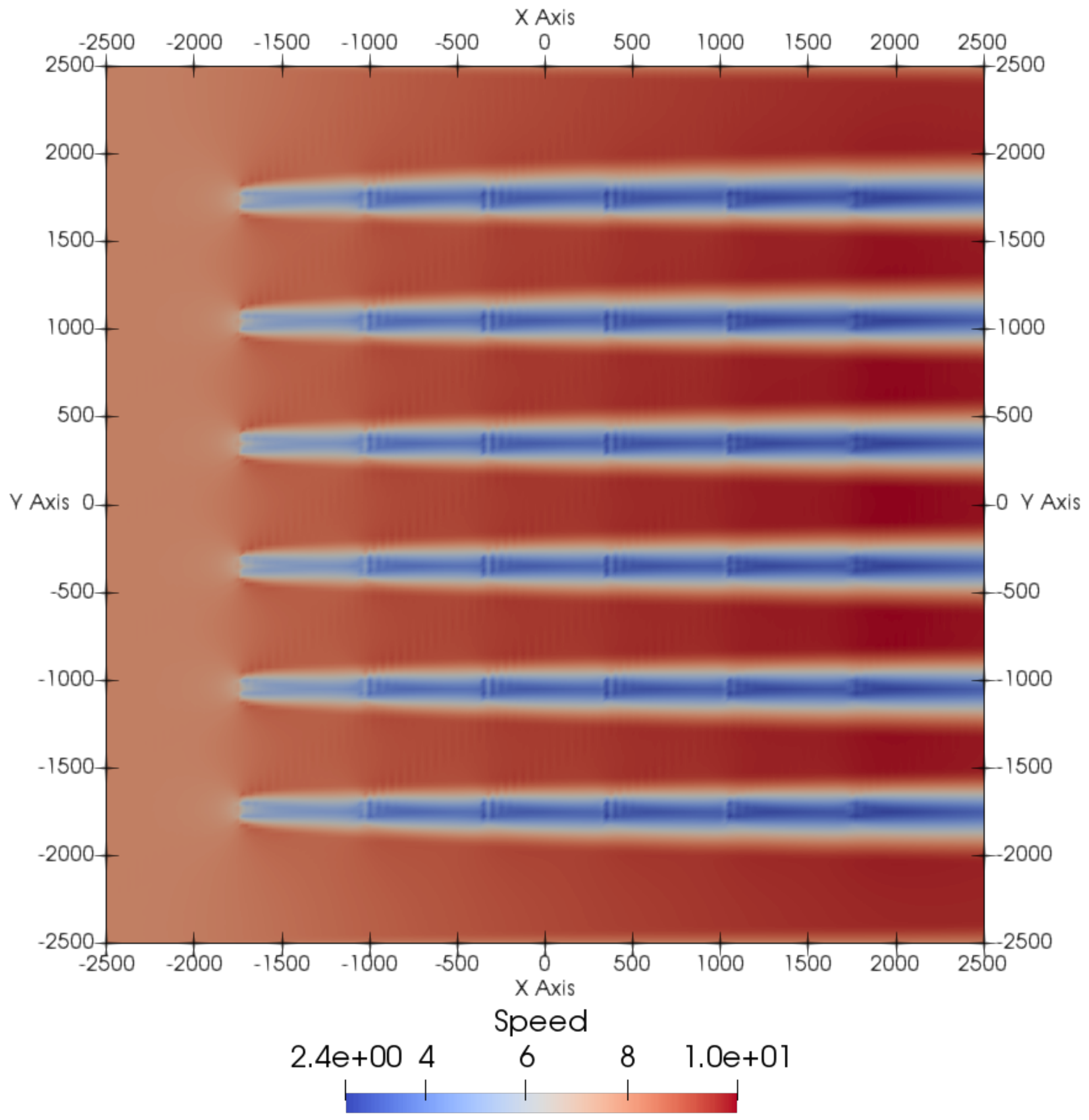
```
solver = df.BuildSolver(params,problem)
```

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

WindSE API

windse.ParameterManager	The ParameterManager controls the handles importing the parameters from the params.yaml file.
windse.DomainManager	The DomainManager submodule contains the various classes used for creating different types of domains
windse.WindFarmManager	The windfarm manager contains everything required to set up a windfarm.
windse.RefinementManager	
windse.FunctionSpaceManager	The FunctionSpaceManager contains all the different types of function spaces required for solve multiple classes of problems.
windse.BoundaryManager	The BoundaryManager submodule contains the classes required for defining the boundary conditions.
windse.ProblemManager	The ProblemManager contains all of the different classes of problems that windse can solve
windse.SolverManager	The SolverManager contains all the different ways to solve problems generated in windse
windse.objective_functions	The objective function live in the windse/objective_functions folder.
windse.OptimizationManager	The OptimizationManager submodule contains all the required function for optimizing via dolfin-adjoint.
windse_driver.driver_functions	

5.1 windse.ParameterManager

The ParameterManager controls the handles importing the parameters from the params.yaml file. These functions don't need to be accessed by the end user.

class windse.ParameterManager.**Parameters**

Bases: dict

Parameters is a subclass of pythons *dict* that adds function specific to windse.

Load(*loc*, *updated_parameters=[]*)
This function loads the parameters from the .yaml file. It should only be assessed once from the `windse.initialize()` function.

Parameters **loc** (*str*) – This string is the location of the .yaml parameters file.

Read()
This function reads the current state of the parameters object and prints it in a easy to read way.

Save(*func*, *filename*, *subfolder=''*, *val=0*, *file=None*, *filetype='default'*)
This function is used to save the various dolfin.Functions created by windse. It should only be accessed internally.

Parameters

- **func** (*dolfin.Function*) – The Function to be saved
- **filename** (*str*) – the name of the function

Keyword Arguments

- **subfolder** (*str*): where to save the files within the output folder
- **n** (*float*): used for saving a series of output. Use n=0 for the first save.

fprint(*string*, *tab=None*, *offset=0*, *special=None*)
This is just a fancy print function that will tab according to where we are in the solve

Parameters **string** (*str*) – the string for printing

Keyword Arguments

- **tab** (*int*): the tab level

5.1.1 Classes

class windse.ParameterManager.**Logger**(*filename*, *std*, *rank*)
Bases: `object`

class windse.ParameterManager.**Parameters**
Bases: `dict`

Parameters is a subclass of pythons *dict* that adds function specific to windse.

Load(*loc*, *updated_parameters=[]*)
This function loads the parameters from the .yaml file. It should only be assessed once from the `windse.initialize()` function.

Parameters **loc** (*str*) – This string is the location of the .yaml parameters file.

Read()
This function reads the current state of the parameters object and prints it in a easy to read way.

Save(*func*, *filename*, *subfolder=''*, *val=0*, *file=None*, *filetype='default'*)
This function is used to save the various dolfin.Functions created by windse. It should only be accessed internally.

Parameters

- **func** (*dolfin.Function*) – The Function to be saved
- **filename** (*str*) – the name of the function

Keyword Arguments

- **subfolder** (*str*): where to save the files within the output folder
- **n** (*float*): used for saving a series of output. Use n=0 for the first save.

fprint (*string, tab=None, offset=0, special=None*)
This is just a fancy print function that will tab according to where we are in the solve

Parameters string (*str*) – the string for printing

Keyword Arguments

- **tab** (*int*): the tab level

5.2 windse.DomainManager

The DomainManager submodule contains the various classes used for creating different types of domains

class windse.DomainManager.**BoxDomain**
Bases: *windse.DomainManager.GenericDomain*

A box domain is simply a 3D rectangular prism. This box is defined by 6 parameters in the param.yaml file.

Example

In the yaml file define:

```
domain:
    #                         # Description            | Units
    x_range: [-2500, 2500] # x-range of the domain | m
    y_range: [-2500, 2500] # y-range of the domain | m
    z_range: [0.04, 630]    # z-range of the domain | m
    nx: 10                  # Number of x-nodes     | -
    ny: 10                  # Number of y-nodes     | -
    nz: 2                   # Number of z-nodes     | -
```

This will produce a box with corner points (-2500,-2500,0.04) to (2500,2500,630). The mesh will have *nx* nodes in the *x*-direction, *ny* in the *y*-direction, and *nz* in the *z*-direction.

class windse.DomainManager.**CircleDomain**
Bases: *windse.DomainManager.GenericDomain*

ADD DOCUMENTATION

class windse.DomainManager.**CylinderDomain**
Bases: *windse.DomainManager.GenericDomain*

A cylinder domain is a cylinder that is centered a c0 and has radius r. This domain is defined by 6 parameters in the param.yaml file. The center of the cylinder is assumed to be the z-axis.

Example

In the yaml file define:

```
domain:
    #                         # Description            | Units
    z_range: [0.04, 630]    # z-range of the domain | m
    radius: 2500            # radius of base circle | m
```

(continues on next page)

(continued from previous page)

```
    nt: 100                 # Number of radial nodes| -
    nz: 10                  # Number of z nodes     | -
```

This will produce a upright cylinder centered at (0.0,0.0) with a radius of 2500 m and extends from z=0.04 to 630 m. The mesh will have *nx* nodes in the *x*-direction, *ny* in the *y*-direction, and *nz* in the *z*-direction.

class `windse.DomainManager.`**`GenericDomain`**

Bases: `object`

A GenericDomain contains on the basic functions required by all domain objects

`Ground`(*x*, *y*, *dx=0*, *dy=0*)

Ground returns the ground height given an (*x*, *y*) coordinate.

Parameters

- **x** (*float/list*) – *x* location within the domain
- **y** (*float/list*) – *y* location within the domain

Returns corresponding z coordinates of the ground.

Return type float/list

`Plot`()

This function plots the domain using matplotlib and saves the output to output/. . . /plots/mesh.pdf

`Save`(*val=0*)

This function saves the mesh and boundary markers to output/. . . /mesh/

`WarpSmooth`(*s*)

This function warps the mesh to shift more cells towards the ground. The cells are shifted based on the function:

$$z_new = z_0 + (z_1 - z_0)\left(\frac{z_old - z_0}{z_1 - z_0}\right)^s.$$

where z_0 is the ground and z_1 is the top of the domain.

Parameters **s** (*float*) – compression strength

`WarpSplit`(*h*, *s*)

This function warps the mesh to shift more cells towards the ground. is achieved by spliting the domain in two and moving the cells so that a percentage of them are below the split.

Parameters

- **h** (*float*) – the height that split occurs
- **s** (*float*) – the percent below split in the range [0,1)

class `windse.DomainManager.`**`ImportedDomain`**

Bases: *`windse.DomainManager.GenericDomain`*

This class generates a domain from imported files. This mesh is defined by 2 parameters in the param.yaml file.

Example

In the yaml file define:

```
domain:
    path: "Mesh_data/"
    filetype: "xml.gz"
```

The supported filetypes are “xml.gz” and “h5”. For “xml.gz” 3 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.** It 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
  2 2 0.2
      1.0
      1.1
      1.2
      2.0
      2.1
      2.2
  ```

class windse.DomainManager.**InterpolatedBoxDomain**
    Bases: *windse.DomainManager.BoxDomain*

class windse.DomainManager.**InterpolatedCylinderDomain**
    Bases: *windse.DomainManager.CylinderDomain*

class windse.DomainManager.**PeriodicDomain**
    Bases: *windse.DomainManager.BoxDomain*

class windse.DomainManager.**RectangleDomain**
    Bases: *windse.DomainManager.GenericDomain*

A rectangle domain is simply a 2D rectangle. This mesh is defined by 4 parameters in the param.yaml file.

Example

In the yaml file define:

```
domain:
    #                         # Description            | Units
    x_range: [-2500, 2500] # x-range of the domain | m
    y_range: [-2500, 2500] # y-range of the domain | m
    nx: 10                    # Number of x-nodes      | -
    ny: 10                    # Number of y-nodes      | -
```

This will produce a rectangle with corner points (-2500,-2500) to (2500,2500). The mesh will have *nx* nodes in the *x*-direction, and *ny* in the *y*-direction.

Todo: Properly implement a RectangleDomain and 2D in general.

5.2.1 Classes

class windse.DomainManager.**BoxDomain**

Bases: *windse.DomainManager.GenericDomain*

A box domain is simply a 3D rectangular prism. This box is defined by 6 parameters in the param.yaml file.

Example

In the yaml file define:

```
domain:
    #                          # Description            | Units
    x_range: [-2500, 2500] # x-range of the domain | m
    y_range: [-2500, 2500] # y-range of the domain | m
    z_range: [0.04, 630]   # z-range of the domain | m
    nx: 10                 # Number of x-nodes     | -
    ny: 10                 # Number of y-nodes     | -
    nz: 2                  # Number of z-nodes     | -
```

This will produce a box with corner points (-2500,-2500,0.04) to (2500,2500,630). The mesh will have *nx* nodes in the *x*-direction, *ny* in the *y*-direction, and *nz* in the *z*-direction.

class windse.DomainManager.**CircleDomain**

Bases: *windse.DomainManager.GenericDomain*

ADD DOCUMENTATION

class windse.DomainManager.**CylinderDomain**

Bases: *windse.DomainManager.GenericDomain*

A cylinder domain is a cylinder that is centered a c0 and has radius r. This domain is defined by 6 parameters in the param.yaml file. The center of the cylinder is assumed to be the z-axis.

Example

In the yaml file define:

```
domain:
    #                          # Description            | Units
    z_range: [0.04, 630]   # z-range of the domain | m
    radius: 2500           # radius of base circle | m
    nt: 100                # Number of radial nodes| -
    nz: 10                 # Number of z nodes     | -
```

This will produce a upright cylinder centered at (0.0,0.0) with a radius of 2500 m and extends from z=0.04 to 630 m. The mesh will have *nx* nodes in the *x*-direction, *ny* in the *y*-direction, and *nz* in the *z*-direction.

class windse.DomainManager.**GenericDomain**

Bases: `object`

A GenericDomain contains on the basic functions required by all domain objects

Ground(*x*, *y*, *dx=0*, *dy=0*)

Ground returns the ground height given an (*x*, *y*) coordinate.

Parameters

- **x** (*float/list*) – *x* location within the domain

- **y** (*float/list*) – *y* location within the domain

Returns corresponding z coordinates of the ground.

Return type float/list

Plot()
This function plots the domain using matplotlib and saves the output to output/. . . /plots/mesh.pdf

Save(*val=0*)
This function saves the mesh and boundary markers to output/. . . /mesh/

WarpSmooth(*s*)
This function warps the mesh to shift more cells towards the ground. The cells are shifted based on the function:

$$z_new = z_0 + (z_1 - z_0)\left(\frac{z_old - z_0}{z_1 - z_0}\right)^s.$$

where z_0 is the ground and z_1 is the top of the domain.

Parameters **s** (*float*) – compression strength

WarpSplit(*h, s*)
This function warps the mesh to shift more cells towards the ground. is achieved by spliting the domain in two and moving the cells so that a percentage of them are below the split.

Parameters

- **h** (*float*) – the height that split occurs
- **s** (*float*) – the percent below split in the range [0,1)

class windse.DomainManager.**ImportedDomain**
Bases: *windse.DomainManager.GenericDomain*

This class generates a domain from imported files. This mesh is defined by 2 parameters in the param.yaml file.

Example

In the yaml file define:

```
domain:
    path: "Mesh_data/"
    filetype: "xml.gz"
```

The supported filetypes are “xml.gz” and “h5”. For “xml.gz” 3 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.** It 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
2 2 0.2
    1.0
```

(continues on next page)

(continued from previous page)

```
        1.1
        1.2
        2.0
        2.1
        2.2
```

class windse.DomainManager.**InterpolatedBoxDomain**

Bases: *windse.DomainManager.BoxDomain*

class windse.DomainManager.**InterpolatedCylinderDomain**

Bases: *windse.DomainManager.CylinderDomain*

class windse.DomainManager.**PeriodicDomain**

Bases: *windse.DomainManager.BoxDomain*

class windse.DomainManager.**RectangleDomain**

Bases: *windse.DomainManager.GenericDomain*

A rectangle domain is simply a 2D rectangle. This mesh is defined by 4 parameters in the param.yaml file.

Example

In the yaml file define:

```
domain:
    #                        # Description            | Units
    x_range: [-2500, 2500]   # x-range of the domain | m
    y_range: [-2500, 2500]   # y-range of the domain | m
    nx: 10                   # Number of x-nodes     | -
    ny: 10                   # Number of y-nodes     | -
```

This will produce a rectangle with corner points (-2500,-2500) to (2500,2500). The mesh will have *nx* nodes in the *x*-direction, and *ny* in the *y*-direction.

Todo: Properly implement a RectangleDomain and 2D in general.

5.2.2 Functions

windse.DomainManager.**Elliptical_Grid**(*x, y, z, radius*)

windse.DomainManager.**FG_Squircular**(*x, y, z, radius*)

windse.DomainManager.**Simple_Stretching**(*x, y, z, radius*)

5.3 windse.WindFarmManager

The windfarm manager contains everything required to set up a windfarm.

class windse.WindFarmManager.**EmptyWindFarm**(*dom*)

Bases: *windse.WindFarmManager.GenericWindFarm*

class `windse.WindFarmManager.`**`GenericWindFarm`**(*dom*)

Bases: `object`

A GenericProblem contains on the basic functions required by all problem objects.

Parameters **dom** (*windse.DomainManager.GenericDomain()*) – a windse domain object.

`CalculateFarmBoundingBox`()

This functions takes into consideration the turbine locations, diameters, and hub heights to create lists that describe the extent of the windfarm. These lists are append to the parameters object.

`CalculateHeights`()

This function calculates the absolute heights of each turbine.

`CreateConstants`()

This functions converts lists of locations and axial inductions into dolfin.Constants. This is useful in optimization.

`CreateLists`()

This function creates lists from single values. This is useful when the params.yaml file defines only one type of turbine.

`DolfinTurbineForce`(*fs*, *mesh*, *inflow_angle=0.0*)

This function creates a turbine force by applying a spacial kernel to each turbine. This kernel is created from the turbines location, yaw, thickness, diameter, and force density. Currently, force density is limit to a scaled version of

$$r\sin(r),$$

where r is the distance from the center of the turbine.

Parameters

- **V** (*dolfin.FunctionSpace*) – The function space the turbine force will use.
- **mesh** (*dolfin.mesh*) – The mesh

Returns the turbine force.

Return type tf (dolfin.Function)

Todo:

- Setup a way to get the force density from file

`PlotFarm`(*show=False*, *filename='wind_farm'*, *power=None*)

This function plots the locations of each wind turbine and saves the output to output/…/plots/

Keyword Arguments

- **show** (*bool*): Default: True, Set False to suppress output but still save.

`SaveActuatorDisks`(*val=0*)

This function saves the turbine force if exists to output/…/functions/

`YawTurbine`(*x*, *x0*, *yaw*)

This function yaws the turbines when creating the turbine force.

Parameters

- **x** (*dolfin.SpatialCoordinate*) – the space variable, x

- **x0** (*list*) – the location of the turbine to be yawed
- **yaw** (*float*) – the yaw value in radians

class windse.WindFarmManager.**GridWindFarm**(*dom*)

Bases: *windse.WindFarmManager.GenericWindFarm*

A GridWindFarm produces turbines on a grid. The params.yaml file determines how this grid is set up.

Example

In the .yaml file you need to define:

```
wind_farm:
    #                     # Description              | Units
    HH: 90                # Hub Height               | m
    RD: 126.0             # Turbine Diameter         | m
    thickness: 10.5       # Effective Thickness      | m
    yaw: 0.0              # Yaw                      | rads
    axial: 0.33           # Axial Induction          | -
    ex_x: [-1500, 1500]   # x-extent of the farm     | m
    ex_y: [-1500, 1500]   # y-extent of the farm     | m
    grid_rows: 6          # Number of rows           | -
    grid_cols: 6          # Number of columns        | -
```

This will produce a 6x6 grid of turbines equally spaced within the region [-1500, 1500]x[-1500, 1500].

Parameters dom (*windse.DomainManager.GenericDomain()*) – a windse domain object.

class windse.WindFarmManager.**ImportedWindFarm**(*dom*)

Bases: *windse.WindFarmManager.GenericWindFarm*

A ImportedWindFarm produces turbines located based on a text file. The params.yaml file determines how this grid is set up.

Example

In the .yaml file you need to define:

```
wind_farm:
    imported: true
    path: "inputs/wind_farm.txt"
```

The “wind_farm.txt” needs to be set up like this:

```
#    x       y      HH           Yaw Diameter Thickness Axial_Induction
200.00 0.0000 80.000  0.0000000000      126       10.5             0.33
800.00 0.0000 80.000  0.0000000000      126       10.5             0.33
```

The first row isn’t necessary. Each row defines a different turbine.

Parameters dom (*windse.DomainManager.GenericDomain()*) – a windse domain object.

class windse.WindFarmManager.**RandomWindFarm**(*dom*)

Bases: *windse.WindFarmManager.GenericWindFarm*

A RandomWindFarm produces turbines located randomly with a defined range. The params.yaml file determines how this grid is set up.

Example

In the .yaml file you need to define:

```
wind_farm:
    #                     # Description              | Units
    HH: 90                # Hub Height               | m
    RD: 126.0             # Turbine Diameter         | m
    thickness: 10.5       # Effective Thickness      | m
    yaw: 0.0              # Yaw                      | rads
    axial: 0.33           # Axial Induction          | -
    ex_x: [-1500, 1500]   # x-extent of the farm     | m
    ex_y: [-1500, 1500]   # y-extent of the farm     | m
    numturbs: 36          # Number of Turbines       | -
    seed: 15              # Random Seed for Numpy    | -
```

This will produce a 36 turbines randomly located within the region [-1500, 1500]x[-1500, 1500]. The seed is optional but useful for reproducing test.

Parameters **dom** (`windse.DomainManager.GenericDomain()`) – a windse domain object.

5.3.1 Classes

class `windse.WindFarmManager.`**`EmptyWindFarm`**(*dom*)
Bases: `windse.WindFarmManager.GenericWindFarm`

class `windse.WindFarmManager.`**`GenericWindFarm`**(*dom*)
Bases: `object`

A GenericProblem contains on the basic functions required by all problem objects.

Parameters **dom** (`windse.DomainManager.GenericDomain()`) – a windse domain object.

`CalculateFarmBoundingBox`()
This functions takes into consideration the turbine locations, diameters, and hub heights to create lists that describe the extent of the windfarm. These lists are append to the parameters object.

`CalculateHeights`()
This function calculates the absolute heights of each turbine.

`CreateConstants`()
This functions converts lists of locations and axial inductions into dolfin.Constants. This is useful in optimization.

`CreateLists`()
This function creates lists from single values. This is useful when the params.yaml file defines only one type of turbine.

`DolfinTurbineForce`(*fs*, *mesh*, *inflow_angle=0.0*)
This function creates a turbine force by applying a spacial kernel to each turbine. This kernel is created from the turbines location, yaw, thickness, diameter, and force density. Currently, force density is limit to

a scaled version of

$$r \sin(r),$$

where r is the distance from the center of the turbine.

Parameters

- **V** (*dolfin.FunctionSpace*) – The function space the turbine force will use.
- **mesh** (*dolfin.mesh*) – The mesh

Returns the turbine force.

Return type tf (dolfin.Function)

Todo:

- Setup a way to get the force density from file

PlotFarm(*show=False, filename='wind_farm', power=None*)
This function plots the locations of each wind turbine and saves the output to output/. . . /plots/

Keyword Arguments

- **show** (*bool*): Default: True, Set False to suppress output but still save.

SaveActuatorDisks(*val=0*)
This function saves the turbine force if exists to output/. . . /functions/

YawTurbine(*x, x0, yaw*)
This function yaws the turbines when creating the turbine force.

Parameters

- **x** (*dolfin.SpatialCoordinate*) – the space variable, x
- **x0** (*list*) – the location of the turbine to be yawed
- **yaw** (*float*) – the yaw value in radians

class windse.WindFarmManager.**GridWindFarm**(*dom*)
Bases: *windse.WindFarmManager.GenericWindFarm*

A GridWindFarm produces turbines on a grid. The params.yaml file determines how this grid is set up.

Example

In the .yaml file you need to define:

```
wind_farm:
    #                      # Description              | Units
    HH: 90                 # Hub Height               | m
    RD: 126.0              # Turbine Diameter         | m
    thickness: 10.5        # Effective Thickness      | m
    yaw: 0.0               # Yaw                      | rads
    axial: 0.33            # Axial Induction          | -
    ex_x: [-1500, 1500]    # x-extent of the farm     | m
    ex_y: [-1500, 1500]    # y-extent of the farm     | m
    grid_rows: 6           # Number of rows           | -
    grid_cols: 6           # Number of columns        | -
```

This will produce a 6x6 grid of turbines equally spaced within the region [-1500, 1500]x[-1500, 1500].

Parameters **dom** (*windse.DomainManager.GenericDomain()*) – a windse domain object.

class windse.WindFarmManager.**ImportedWindFarm**(*dom*)

Bases: *windse.WindFarmManager.GenericWindFarm*

A ImportedWindFarm produces turbines located based on a text file. The params.yaml file determines how this grid is set up.

Example

In the .yaml file you need to define:

```
wind_farm:
    imported: true
    path: "inputs/wind_farm.txt"
```

The "wind_farm.txt" needs to be set up like this:

```
#    x       y      HH            Yaw Diameter Thickness Axial_Induction
200.00 0.0000 80.000  0.0000000000      126       10.5            0.33
800.00 0.0000 80.000  0.0000000000      126       10.5            0.33
```

The first row isn't necessary. Each row defines a different turbine.

Parameters **dom** (*windse.DomainManager.GenericDomain()*) – a windse domain object.

class windse.WindFarmManager.**RandomWindFarm**(*dom*)

Bases: *windse.WindFarmManager.GenericWindFarm*

A RandomWindFarm produces turbines located randomly with a defined range. The params.yaml file determines how this grid is set up.

Example

In the .yaml file you need to define:

```
wind_farm:
    #                      # Description              | Units
    HH: 90                 # Hub Height               | m
    RD: 126.0              # Turbine Diameter         | m
    thickness: 10.5        # Effective Thickness      | m
    yaw: 0.0               # Yaw                      | rads
    axial: 0.33            # Axial Induction          | -
    ex_x: [-1500, 1500]    # x-extent of the farm     | m
    ex_y: [-1500, 1500]    # y-extent of the farm     | m
    numturbs: 36           # Number of Turbines       | -
    seed: 15               # Random Seed for Numpy    | -
```

This will produce a 36 turbines randomly located within the region [-1500, 1500]x[-1500, 1500]. The seed is optional but useful for reproducing test.

Parameters **dom** (*windse.DomainManager.GenericDomain()*) – a windse domain object.

5.4 windse.RefinementManager

5.4.1 Functions

`windse.RefinementManager.`**`CreateRefinementList`**`(`*dom*, *farm*, *refine_params*`)`

`windse.RefinementManager.`**`RefineMesh`**`(`*dom*, *farm*`)`

`windse.RefinementManager.`**`WarpMesh`**`(`*dom*`)`

5.5 windse.FunctionSpaceManager

The FunctionSpaceManager contains all the different types of function spaces required for solve multiple classes of problems.

`class` `windse.FunctionSpaceManager.`**`LinearFunctionSpace`**`(`*dom*`)`
Bases: *`windse.FunctionSpaceManager.GenericFunctionSpace`*

The LinearFunctionSpace is made up of a vector function space for velocity and a scaler space for pressure. Both spaces are “CG1” or Linear Lagrange elements.

`class` `windse.FunctionSpaceManager.`**`TaylorHoodFunctionSpace`**`(`*dom*`)`
Bases: *`windse.FunctionSpaceManager.GenericFunctionSpace`*

The TaylorHoodFunctionSpace is made up of a vector function space for velocity and a scalar space for pressure. The velocity function space is piecewise quadratic and the pressure function space is piecewise linear.

5.5.1 Classes

`class` `windse.FunctionSpaceManager.`**`GenericFunctionSpace`**`(`*dom*`)`
Bases: `object`

`class` `windse.FunctionSpaceManager.`**`LinearFunctionSpace`**`(`*dom*`)`
Bases: *`windse.FunctionSpaceManager.GenericFunctionSpace`*

The LinearFunctionSpace is made up of a vector function space for velocity and a scaler space for pressure. Both spaces are “CG1” or Linear Lagrange elements.

`class` `windse.FunctionSpaceManager.`**`TaylorHoodFunctionSpace`**`(`*dom*`)`
Bases: *`windse.FunctionSpaceManager.GenericFunctionSpace`*

The TaylorHoodFunctionSpace is made up of a vector function space for velocity and a scalar space for pressure. The velocity function space is piecewise quadratic and the pressure function space is piecewise linear.

5.6 windse.BoundaryManager

The BoundaryManager submodule contains the classes required for defining the boundary conditions.

`class` `windse.BoundaryManager.`**`PowerInflow`**`(`*dom*, *fs*, *farm*`)`
Bases: *`windse.BoundaryManager.GenericBoundary`*

PowerInflow creates a set of boundary conditions where the x-component of velocity follows a power law. Currently the function is

$$u_x = 8.0 \left(\frac{z - z_0}{z_1 - z_0} \right)^{0.15}.$$

where z_0 is the ground and z_1 is the top of the domain.

Parameters

- **dom** (*windse.DomainManager.GenericDomain*) – A windse domain object.
- **fs** (*windse.FunctionSpaceManager.GenericFunctionSpace*) – A windse function space object

Todo:

- Make the max velocity an input
- Make the power an input

5.6.1 Classes

class windse.BoundaryManager.**GenericBoundary**(*dom*, *fs*, *farm*)

Bases: `object`

SaveHeight(*val=0*)

This function saves the turbine force if exists to output/. . . /functions/

SaveInitialGuess(*val=0*)

This function saves the turbine force if exists to output/. . . /functions/

class windse.BoundaryManager.**LogLayerInflow**(*dom*, *fs*, *farm*)

Bases: *windse.BoundaryManager.GenericBoundary*

class windse.BoundaryManager.**PowerInflow**(*dom*, *fs*, *farm*)

Bases: *windse.BoundaryManager.GenericBoundary*

PowerInflow creates a set of boundary conditions where the x-component of velocity follows a power law. Currently the function is

$$u_x = 8.0 \left(\frac{z - z_0}{z_1 - z_0} \right)^{0.15}.$$

where z_0 is the ground and z_1 is the top of the domain.

Parameters

- **dom** (*windse.DomainManager.GenericDomain*) – A windse domain object.
- **fs** (*windse.FunctionSpaceManager.GenericFunctionSpace*) – A windse function space object

Todo:

- Make the max velocity an input
- Make the power an input

class `windse.BoundaryManager.`**`TurbSimInflow`**(*dom*, *fs*, *farm*)
Bases: `windse.BoundaryManager.LogLayerInflow`

class `windse.BoundaryManager.`**`UniformInflow`**(*dom*, *fs*, *farm*)
Bases: `windse.BoundaryManager.GenericBoundary`

5.7 windse.ProblemManager

The ProblemManager contains all of the different classes of problems that windse can solve

class `windse.ProblemManager.`**`GenericProblem`**(*domain*, *windfarm*, *function_space*, *boundary_data*)
Bases: `object`

A GenericProblem contains on the basic functions required by all problem objects.

Parameters

- **domain** (`windse.DomainManager.GenericDomain()`) – a windse domain object.
- **windfarm** (`windse.WindFarmManager.GenericWindFarmm()`) – a windse windfarm object.
- **function_space** (`windse.FunctionSpaceManager.GenericFunctionSpace()`) – a windse function space object.
- **boundary_conditions** (`windse.BoundaryManager.GenericBoundary()`) – a windse boundary object.

`ChangeWindAngle`(*inflow_angle*)
This function recomputes all necessary components for a new wind direction

Parameters **inflow_angle** (*`float`*) – The new wind angle in radians

class `windse.ProblemManager.`**`IterativeSteady`**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)
Bases: `windse.ProblemManager.GenericProblem`

The IterativeSteady sets up everything required for solving Navier-Stokes using the SIMPLE algorithm

Parameters

- **domain** (`windse.DomainManager.GenericDomain()`) – a windse domain object.
- **windfarm** (`windse.WindFarmManager.GenericWindFarmm()`) – a windse windfarm object.
- **function_space** (`windse.FunctionSpaceManager.GenericFunctionSpace()`) – a windse function space object.
- **boundary_conditions** (`windse.BoundaryManager.GenericBoundary()`) – a windse boundary object.

class `windse.ProblemManager.`**`StabilizedProblem`**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)
Bases: `windse.ProblemManager.GenericProblem`

The StabilizedProblem setup everything required for solving Navier-Stokes with a stabilization term

Parameters

- **domain** (`windse.DomainManager.GenericDomain()`) – a windse domain object.

- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

class windse.ProblemManager.**TaylorHoodProblem**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)

Bases: *windse.ProblemManager.GenericProblem*

The TaylorHoodProblem sets up everything required for solving Navier-Stokes

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

class windse.ProblemManager.**UnsteadyProblem**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)

Bases: *windse.ProblemManager.GenericProblem*

The UnsteadyProblem sets up everything required for solving Navier-Stokes using a fractional-step method with an adaptive timestep size

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

5.7.1 Classes

class windse.ProblemManager.**GenericProblem**(*domain*, *windfarm*, *function_space*, *boundary_data*)

Bases: object

A GenericProblem contains on the basic functions required by all problem objects.

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.

- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

ChangeWindAngle(*inflow_angle*)
This function recomputes all necessary components for a new wind direction

Parameters **inflow_angle** (*float*) – The new wind angle in radians

class windse.ProblemManager.**IterativeSteady**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)
Bases: *windse.ProblemManager.GenericProblem*

The IterativeSteady sets up everything required for solving Navier-Stokes using the SIMPLE algorithm

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

class windse.ProblemManager.**StabilizedProblem**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)
Bases: *windse.ProblemManager.GenericProblem*

The StabilizedProblem setup everything required for solving Navier-Stokes with a stabilization term

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

class windse.ProblemManager.**TaylorHoodProblem**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)
Bases: *windse.ProblemManager.GenericProblem*

The TaylorHoodProblem sets up everything required for solving Navier-Stokes

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.

- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

class windse.ProblemManager.**UnsteadyProblem**(*domain*, *windfarm*, *function_space*, *boundary_conditions*)

Bases: *windse.ProblemManager.GenericProblem*

The UnsteadyProblem sets up everything required for solving Navier-Stokes using a fractional-step method with an adaptive timestep size

Parameters

- **domain** (*windse.DomainManager.GenericDomain()*) – a windse domain object.
- **windfarm** (windse.WindFarmManager.GenericWindFarmm()) – a windse windfarm object.
- **function_space** (*windse.FunctionSpaceManager.GenericFunctionSpace()*) – a windse function space object.
- **boundary_conditions** (*windse.BoundaryManager.GenericBoundary()*) – a windse boundary object.

5.8 windse.SolverManager

The SolverManager contains all the different ways to solve problems generated in windse

class windse.SolverManager.**GenericSolver**(*problem*)

Bases: object

A GenericSolver contains on the basic functions required by all solver objects.

ChangeWindAngle(*inflow_angle*)

This function recomputes all necessary components for a new wind direction

Parameters **theta** (*float*) – The new wind angle in radians

ChangeWindSpeed(*inflow_speed*)

This function recomputes all necessary components for a new wind direction

Parameters **theta** (*float*) – The new wind angle in radians

Save(*val=0*)

This function saves the mesh and boundary markers to output/. . . /solutions/

class windse.SolverManager.**IterativeSteadySolver**(*problem*)

Bases: *windse.SolverManager.GenericSolver*

This solver is for solving the iterative steady state problem

Parameters **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.

Solve()

This solves the problem setup by the problem object.

class windse.SolverManager.**MultiAngleSolver**(*problem*)

Bases: *windse.SolverManager.SteadySolver*

This solver will solve the problem using the steady state solver for every angle in angles.

Parameters

- **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.
- **angles** (*list*) – A list of wind inflow directions.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**SteadySolver**(*problem*)
Bases: *windse.SolverManager.GenericSolver*

This solver is for solving the steady state problem

Parameters **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**TimeSeriesSolver**(*problem*)
Bases: *windse.SolverManager.SteadySolver*

This solver will solve the problem using the steady state solver for every angle in angles.

Parameters

- **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.
- **angles** (*list*) – A list of wind inflow directions.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**UnsteadySolver**(*problem*)
Bases: *windse.SolverManager.GenericSolver*

This solver is for solving an unsteady problem. As such, it contains additional time-stepping features and functions not present in other solvers. This solver can only be used if an unsteady problem has been specified in the input file.

Parameters **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.

5.8.1 Classes

class windse.SolverManager.**GenericSolver**(*problem*)
Bases: object

A GenericSolver contains on the basic functions required by all solver objects.

ChangeWindAngle(*inflow_angle*)
This function recomputes all necessary components for a new wind direction

Parameters **theta** (*float*) – The new wind angle in radians

ChangeWindSpeed(*inflow_speed*)
This function recomputes all necessary components for a new wind direction

Parameters **theta** (*float*) – The new wind angle in radians

Save(*val=0*)
This function saves the mesh and boundary markers to output/…/solutions/

class windse.SolverManager.**IterativeSteadySolver**(*problem*)
Bases: *windse.SolverManager.GenericSolver*

This solver is for solving the iterative steady state problem

Parameters **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**MultiAngleSolver**(*problem*)
Bases: *windse.SolverManager.SteadySolver*

This solver will solve the problem using the steady state solver for every angle in angles.

Parameters

- **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.
- **angles** (*list*) – A list of wind inflow directions.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**SteadySolver**(*problem*)
Bases: *windse.SolverManager.GenericSolver*

This solver is for solving the steady state problem

Parameters **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**TimeSeriesSolver**(*problem*)
Bases: *windse.SolverManager.SteadySolver*

This solver will solve the problem using the steady state solver for every angle in angles.

Parameters

- **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.
- **angles** (*list*) – A list of wind inflow directions.

Solve()
This solves the problem setup by the problem object.

class windse.SolverManager.**UnsteadySolver**(*problem*)
Bases: *windse.SolverManager.GenericSolver*

This solver is for solving an unsteady problem. As such, it contains additional time-stepping features and functions not present in other solvers. This solver can only be used if an unsteady problem has been specified in the input file.

Parameters **problem** (*windse.ProblemManager.GenericProblem()*) – a windse problem object.

5.9 windse.objective_functions

The objective function live in the windse/objective_functions folder. These functions should be called using the dictionary objective_funcs[<name>](solver, *args, **kwargs), where <name> is the function name.

5.9.1 Functions

`windse.objective_functions.`**`2d_power`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

The "2d_power" objective function calculates the power using actuator disks, by computing the integral of turbine force dotted with velocity. Additionally, some modification are made to account for the fact the simulation is 2D.

`windse.objective_functions.`**`KE_entrainment`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

The "KE_entrainment" objective function computed the vertical kinetic entrainment behind a single turbine

Keyword Arguments **`ke_location`** – location of measurement, hub, rotor, tip (only rotor works for now)

`windse.objective_functions.`**`alm_power`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

The "alm_power" objective function computes the power using actuator lines by dotting the turbine force in the rotor plane with the moment arm of the turbine blade multiplied by angular velocity. Can be used for multiple turbines.

Keyword Arguments **`alm_power_type`** – real or fake; real - dotting the turbine force with the moment arm of the turbine blade fake - simply multiply the turbine force by the velocity

`windse.objective_functions.`**`cyld_kernel`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

This is a blockage metric that measures the velocity within a Gaussian cylinder located upstream from each turbine and aligned with the rotor's rotational axis or overhead from each turbine and aligned with the mast. The cylindrical Gaussian field is formed by intersecting a radial Gaussian and a streamwise Gaussian.

Keyword Arguments

- **`type`** – The orientation of the Gaussian cylinder, "upstream" for Gaussians shifted to measure the velocity directly upstream from each turbine, "above" to orient the Gaussians over the top of each turbine.
- **`radius`** – The radius of the cylinder, expressed in units of rotor diameter (RD). The default of 0.5 sets the radius to match the turbine radius, 0.5*RD
- **`length`** – The length of the cylinder, expressed in units of rotor diameter (RD). The default of 3.0 means the measurement volume has an axial length of 3*RD.
- **`sharpness`** – The sharpness value controls the severity with which the Gaussian field drops to zero outside the volume of the Gaussian cylinder. The sharpness value *must* be an even number. Smaller values result in smoother transitions over longer length scales, larger values result in more abrupt transitions. For very large values, the transition becomes a near step change which may not have valid values of the derivative. The default setting of 6 is a good starting point.

`windse.objective_functions.`**`mean_point_blockage`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

This is a simple blockage metric that evaluates the velocity deficit at a single location above the mean location of all turbine in the farm.

Keyword Arguments **`z_value`** – z location to evaluate `windse.objective_functions.`**`plane_blockage`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

This is a simple blockage metric that integrates the velocity deficit in a plane in front of or above the farm.

Keyword Arguments

- **`axis`** – the orientation of the plane, “z” for above, “x” for in front
- **`thickness`** – how thick of a plane to integrate over
- **`center`** – distance along the axis where the plane is centered

`windse.objective_functions.`**`point_blockage`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

This is a simple blockage metric that evaluates the velocity deficit at a single location in the farm.

Keyword Arguments **`location`** – where the deficit is evaluated

`windse.objective_functions.`**`power`**(*solver*, *inflow_angle=0.0*, *first_call=False*, *annotate=True*, ***kwargs*)

The “power” objective function calculates the power using actuator disks, by computing the integral of turbine force dotted with velocity.

`windse.objective_functions.`**`wake_center`**(*solver*, *inflow_angle=0.0*, *first_call=False*, ***kwargs*)

The “wake_center” objective function computes the wake n rotor diameters downstream from a single turbine. This calculation is perform by centering a cylinder oriented along the streamwise direction downstream and calculating the center of mass of the velocity deficit, or centroid.

Keyword Arguments

- **`wake_RD`** – Number of rotor diameters downstream where the centroid will be computed
- **`wake_length`** – The streamwise length for the area of integration (not used)
- **`wake_radius`** – The radius of the cylinder (not used)

5.10 windse.OptimizationManager

The OptimizationManager submodule contains all the required function for optimizing via dolfin-adjoint. To use dolfin-adjoin set:

```
general:
    dolfin_adjoint: True
```

in the param.yaml file.

Todo:

- Read through an update the docstrings for these functions.
- Create specific optimization classes.

`class` `windse.OptimizationManager.`**`ConsComp`**(***kwargs*)

Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

OpenMDAO component to wrap the constraint computation.

A small wrapper used on the fenics methods for computing constraint and Jacobian values using the OpenMDAO syntax.

compute (*inputs*, *outputs*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

compute_partials (*inputs*, *partials*)
Compute sub-jacobian parts. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **partials** (*Jacobian*) – Sub-jac components written to partials[output_name, input_name]..
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.

initialize ()
Perform any one-time initialization run at instantiation.

setup ()
Declare inputs and outputs.

Available attributes: name pathname comm options

class windse.OptimizationManager.**ObjComp** (***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

OpenMDAO component to wrap the objective computation from dolfin.

Specifically, we use the J and dJ (function and Jacobian) methods to compute the function value and derivative values as needed by the OpenMDAO optimizers.

compute (*inputs*, *outputs*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

compute_partials (*inputs*, *partials*)
Compute sub-jacobian parts. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].

- **partials** (*Jacobian*) – Sub-jac components written to partials[output_name, input_name]..
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

class windse.OptimizationManager.**Optimizer**(*solver*)
Bases: object

A GenericProblem contains on the basic functions required by all problem objects.

Parameters dom (*windse.DomainManager.GenericDomain()*) – a windse domain object.

Gradient()
Returns a gradient of the objective function

windse.OptimizationManager.**gather**(*m*)
Helper function to gather constraint Jacobians. Adapated from fenics.

windse.OptimizationManager.**om_wrapper**(*J*, *initial_DVs*, *dJ*, *H*, *bounds*, ***kwargs*)
Custom optimization wrapper to use OpenMDAO optimizers with dolfin-adjoint.

Follows the API as defined by dolfin-adjoint.

Parameters

- **J** (*object*) – Function to compute the model analysis value at a design point.
- **initial_DVs** (*array*) – The initial design variables so we can get the array sizing correct for the OpenMDAO implementation.
- **dJ** (*object*) – Function to compute the Jacobian at a design point.
- **H** (*object*) – Function to compute the Hessian at a design point (not used).
- **bounds** (*array*) – Array of lower and upper bound values for the design variables.

Returns DVs – The optimal design variable values.

Return type array

5.10.1 Classes

class windse.OptimizationManager.**ConsComp**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

OpenMDAO component to wrap the constraint computation.

A small wrapper used on the fenics methods for computing constraint and Jacobian values using the OpenMDAO syntax.

compute(*inputs*, *outputs*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

compute_partials(*inputs*, *partials*)
Compute sub-jacobian parts. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **partials** (*Jacobian*) – Sub-jac components written to partials[output_name, input_name]..
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

class windse.OptimizationManager.**MinimumDistanceConstraint**(*m_pos*, *min_distance=200*)
Bases: object

class windse.OptimizationManager.**ObjComp**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

OpenMDAO component to wrap the objective computation from dolfin.

Specifically, we use the J and dJ (function and Jacobian) methods to compute the function value and derivative values as needed by the OpenMDAO optimizers.

compute(*inputs*, *outputs*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

compute_partials(*inputs*, *partials*)
Compute sub-jacobian parts. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **partials** (*Jacobian*) – Sub-jac components written to partials[output_name, input_name]..

- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

class windse.OptimizationManager.**Optimizer**(*solver*)
Bases: object

A GenericProblem contains on the basic functions required by all problem objects.

Parameters **dom** (*windse.DomainManager.GenericDomain()*) – a windse domain object.

Gradient()
Returns a gradient of the objective function

5.10.2 Functions

windse.OptimizationManager.**gather**(*m*)
Helper function to gather constraint Jacobians. Adapated from fenics.

windse.OptimizationManager.**om_wrapper**(*J*, *initial_DVs*, *dJ*, *H*, *bounds*, ***kwargs*)
Custom optimization wrapper to use OpenMDAO optimizers with dolfin-adjoint.

Follows the API as defined by dolfin-adjoint.

Parameters

- **J** (*object*) – Function to compute the model analysis value at a design point.
- **initial_DVs** (*array*) – The initial design variables so we can get the array sizing correct for the OpenMDAO implementation.
- **dJ** (*object*) – Function to compute the Jacobian at a design point.
- **H** (*object*) – Function to compute the Hessian at a design point (not used).
- **bounds** (*array*) – Array of lower and upper bound values for the design variables.

Returns **DVs** – The optimal design variable values.

Return type array

5.11 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()

5.11.1 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 6

Indices and tables

- genindex
- modindex

Python Module Index

Index