OpenTidalFarm

# OpenTidalFarm Documentation

***Release 1.5***

**The OpenTidalFarm team**

January 19, 2015

Contents

OpenTidalFarm is an open-source optimisation software for tidal turbine farms.

The positioning of the turbines in a tidal farm is a crucial decision. Simulations show that the optimal positioning can increase the power generation of the farm by up to 50% and can therefore determine the viability of a project. However, finding the optimal layout is a difficult process due to the complex flow interactions. OpenTidalFarm solves this problem by applying an efficient optimisation algorithm onto a accurate flow prediction model.

Following presentation gives a quick introduction to OpenTidalFarm: OpenTidalFarm

To download the source code or to report issues visit the GitHub page

CHAPTER 1

# Contents

## 1.1 Installation

It is recommended to use OpenTidalFarm on the Ubuntu operating system.

### 1.1.1 Quick install

Install using git:

```
git clone git@github.com:OpenTidalFarm/OpenTidalFarm.git
cd OpenTidalFarm
git co opentidalfarm-1.5
git submodule init
git submodule update
python setup.py install
```

Install using pip:

```
pip install git+git://github.com/OpenTidalFarm/OpenTidalFarm.git@opentidalfarm-1.5
```

Test the installation with

```
python -c "import opentidalfarm"
```

If no errors occur, your installation was succesfull.

### 1.1.2 Dependencies

OpenTidalFarm 1.5 depends on following packages:

- FEniCS 1.5 (Follow the Ubuntu PPA installation)
- dolfin-adjoint 1.5 (Follow the Ubuntu PPA installation)
- SciPy >=0.11 - e.g. with:

  ```
  pip install scipy
  ```

- Uptide

  ```
  pip install git+git://github.com/stephankramer/uptide.git
  ```

- UTM - e.g. with:

```
pip install utm
```

## 1.2 Features

- High resolution shallow water model for accurate flow prediction.
- Arbitrary shoreline data and bathymetry support.
- Optimise the turbine position and size to maximise the total farm power output.
- Site constraints / minimum distance between turbines.
- Optimisation of up to hundreds of turbines.
- Checkpoint support to restart optimisation.

## 1.3 Examples

### 1.3.1 Sinusoidal wave in a channel

#### Introduction

This example simulates the flow in a channel with oscillating velocity in-/outflow on the west boundary, fixed surface height on the east boundary, and free-slip flow on the north and south boundaries.

The shallow water equations to be solved are

$$\frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u + g \nabla \eta + \frac{c_b}{H} \|u\| u = 0,$$
$$\frac{\partial \eta}{\partial t} + \nabla \cdot (H u) = 0,$$

where

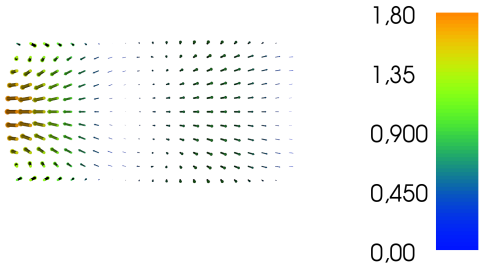
- $u$ is the velocity,
- $\eta$ is the free-surface displacement,
- $H = \eta + h$ is the total water depth where $h$ is the water depth at rest,
- $c_b$ is the (quadratic) natural bottom friction coefficient,
- $\nu$ is the viscosity coefficient,
- $g$ is the gravitational constant.

The boundary conditions are:

$$u = \begin{pmatrix} \sin(\frac{\pi t}{5}) \frac{y(50-y)}{625} \\ 0 \end{pmatrix} \quad \text{on } \Gamma_1,$$
$$\eta = 0 \quad \text{on } \Gamma_2,$$
$$u \cdot n = 0 \quad \text{on } \Gamma_3,$$

where $n$ is the normal vector pointing outside the domain, $\Gamma_1$ is the west boundary of the channel, $\Gamma_2$ is the east boundary of the channel, and $\Gamma_3$ is the north and south boundaries of the channel.

After a few timesteps the solution should looks like this:




### Implementation

We begin with importing the OpenTidalFarm module.

```
from opentidalfarm import *
```

Next we get the default parameters of a shallow water problem and configure it to our needs.

```
prob_params = SWProblem.default_parameters()
```

First we define the computational domain. For a simple channel, we can use the `RectangularDomain` class:

```
domain = RectangularDomain(x0=0, y0=0, x1=100, y1=50, nx=20, ny=10)
```

The boundary of the domain is marked with integers in order to specify different boundary conditions on different parts of the domain. You can plot and inspect the boundary ids with:

```
plot(domain.facet_ids)
```

Once the domain is created we attach it to the problem parameters:

```
prob_params.domain = domain
```

Next we specify boundary conditions. For time-dependent boundary condition use a parameter named $t$ in the `dolfin.Expression` and it will be automatically be updated to the current timelevel during the solve.

```
bcs = BoundaryConditionSet()
u_expr = Expression(("sin(pi*t/5)*x[1]*(50-x[1])/625", "0"), t=Constant(0))
bcs.add_bc("u", u_expr, facet_id=1)
bcs.add_bc("eta", Constant(0), facet_id=2)
```

Free-slip boundary conditions need special attention. The boundary condition type *weak_dirichlet* enforces the boundary value *only* in the *normal* direction of the boundary. Hence, a zero weak Dirichlet boundary condition gives us free-slip, while a zero *strong_dirichlet* boundary condition would give us no-slip.

```
bcs.add_bc("u", Constant((0, 0)), facet_id=3, bctype="strong_dirichlet")
```

Again we attach boundary conditions to the problem parameters:

```
prob_params.bcs = bcs
```

The other parameters are straight forward:

```
# Equation settings
prob_params.viscosity = Constant(30)
prob_params.depth = Constant(20)
prob_params.friction = Constant(0.0)
# Temporal settings
prob_params.theta = Constant(0.5)
```

```
prob_params.start_time = Constant(0)
prob_params.finish_time = Constant(500)
prob_params.dt = Constant(0.5)
# The initial condition consists of three components: u_x, u_y and eta
# Note that we do not set all components to zero, as some components of the
# Jacobian of the quadratic friction term is non-differentiable.
prob_params.initial_condition = Constant((DOLFIN_EPS, 0, 0))
```

Here we only set the necessary options. A full option list with its current values can be viewed with:

```
print prob_params
```

Once the parameter have been set, we create the shallow water problem:

```
problem = SWProblem(prob_params)
```

Next we create a shallow water solver. Here we choose to solve the shallow water equations in its fully coupled form. Again, we first ask for the default parameters, adjust them to our needs and then create the solver object.

```
sol_params = CoupledSWSolver.default_parameters()
sol_params.dump_period = -1
solver = CoupledSWSolver(problem, sol_params)
```

Now we are ready to solve the problem.

```
for s in solver.solve():
    print "Computed solution at time %f" % s["time"]
    plot(s["state"])
interactive()  # Hold the plot until the user presses q.
```

The inner part of the loop is executed for each timestep. The variable `s` is a dictionary and contains information like the current timelevel, the velocity and free-surface functions.

The example code can be found in `examples/channel-simulation/` in the `OpenTidalFarm` source tree, and executed as follows:

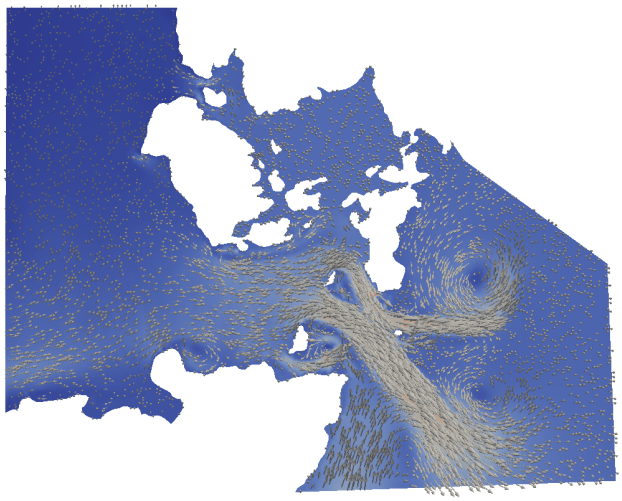
```
$ python channel-simulation.py
```

### 1.3.2 Tidal simulation in the Orkney island

#### Introduction

This example demonstrates how OpenTidalFarm can be used for simulating the tides in a realistic domain.

We will be simulating the tides in the Pentland Firth, Scotland for 12.5 hours, starting at 14:40 am on the 18.9.2001. The flow result at the end of the simulation looks like:



This example requires some large data files, that must be downloaded separately by calling in the source code directory:

```
git submodule init
git submodule update
```

### Implementation

We begin with importing the OpenTidalFarm module.

```
from opentidalfarm import *
```

We also need the datetime module for the tidal forcing.

```
import datetime
```

Next we define the UTM zone and band, as we we will need it multiple times later on.

```
utm_zone = 30
utm_band = 'V'
```

Next we create shallow water problem and attach the domain and boundary conditions

```
prob_params = SWProblem.default_parameters()
```

We load the mesh in UTM coordinates, and boundary ids from file

```
domain = FileDomain("../data/meshes/orkney/orkney_utm.xml")
prob_params.domain = domain
```

The mesh and boundary ids can be visualised with

```
#plot(domain.facet_ids, interactive=True)
```

Next we specify boundary conditions. We apply tidal boundary forcing, by using the `TidalForcing` class.

```
eta_expr = TidalForcing(grid_file_name='../data/netcdf/gridES2008.nc',
                        data_file_name='../data/netcdf/hf.ES2008.nc',
                        ranges=((-4.0,0.0), (58.0,61.0)),
                        utm_zone=utm_zone,
                        utm_band=utm_band,
                        initial_time=datetime.datetime(2001, 9, 18, 10, 40),
```

```
                        constituents=['Q1', 'O1', 'P1', 'K1', 'N2', 'M2', 'S2', 'K2'])

bcs = BoundaryConditionSet()
bcs.add_bc("eta", eta_expr, facet_id=1)
bcs.add_bc("eta", eta_expr, facet_id=2)
```

The free-slip boundary conditions are a special case. The boundary condition type *weak_dirichlet* enforces the boundary value *only* in the *normal* direction of the boundary. Hence, a zero weak Dirichlet boundary condition gives us free-slip, while a zero *strong_dirichlet* boundary condition would give us no-slip.

```
bcs.add_bc("u", Constant((0, 0)), facet_id=3, bctype="strong_dirichlet")
prob_params.bcs = bcs
```

Next we load the bathymetry from the NetCDF file.

```
bathy_expr = BathymetryDepthExpression('../data/netcdf/bathymetry.nc',
        utm_zone=utm_zone, utm_band=utm_band, domain=domain.mesh)
prob_params.depth = bathy_expr
```

The bathymetry can be visualised with

```
#plot(bathy_expr, mesh=domain.mesh, title="Bathymetry", interactive=True)
```

Equation settings

For stability reasons, we want to increase the viscosity at the inflow and outflow boundary conditions. For that, we read in a precomputed function (generated by `'compute_distance'`) and

```
V = FunctionSpace(domain.mesh, "CG", 1)
dist = Function(V)
File("dist.xml") >> dist
```

With that we can define an expression that evaluates to a nu_inside value inside the domain and a nu_outside value near the in/outflow boundary.

```
class ViscosityExpression(Expression):
    def __init__(self, dist_function, dist_threshold, nu_inside, nu_boundary):
        self.dist_function = dist_function
        self.nu_inside = nu_inside
        self.nu_boundary = nu_boundary
        self.dist_threshold = dist_threshold

    def eval(self, value, x):
        if self.dist_function(x) > self.dist_threshold:
            value[0] = self.nu_inside
        else:
            value[0] = self.nu_boundary
```

Finally, we interpolate this expression to a piecewise discontinuous, constant function and attach it as the viscosity value to the shallow water problem.

```
W = FunctionSpace(domain.mesh, "DG", 0)
nu = ViscosityExpression(dist, dist_threshold=1000, nu_inside=10., nu_boundary=1e3)
nu_func = interpolate(nu, W)
prob_params.viscosity = nu_func
```

The other parameters are set as usual.

```
prob_params.friction = Constant(0.0025)
# Temporal settings
```

```
prob_params.start_time = Constant(0)
prob_params.finish_time = Constant(12.5*60*60)
prob_params.dt = Constant(5*60)
# The initial condition consists of three components: u_x, u_y and eta.
# Note that we set the velocity components to a small positive number, as some
# components of the Jacobian of the quadratic friction term is
# non-differentiable.
prob_params.initial_condition = Constant((DOLFIN_EPS, DOLFIN_EPS, 1))

# Now we can create the shallow water problem
problem = SWProblem(prob_params)

# Next we create a shallow water solver. Here we choose to solve the shallow
# water equations in its fully coupled form:
sol_params = CoupledSWSolver.default_parameters()
solver = CoupledSWSolver(problem, sol_params)
```

Now we are ready to solve

```
f_state = File("results/state.pvd")

timer = Timer('')
for s in solver.solve():
    t = float(s["time"])
    log(INFO, "Computed solution at time %f in %f s." % (t, timer.stop()))
    f_state << (s["state"], t)
    timer.start()
```

The code for this example can be found in `examples/tidal-simulation/` in the `OpenTidalFarm` source tree, and executed as follows:

```
$ mpirun -n 4 python orkney-coupled.py
```

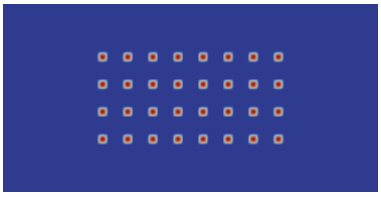
where 4 should be replaced by the number of CPU cores available.

### 1.3.3 Farm layout optimization

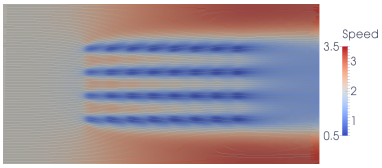
This demo optimizes the position of 32 turbines in a tidal farm within a channel. The goal of the optimization is to maximise the farm's energy extraction. The rectangular channel is 640 m x 320 m large. The farm area is in the channel center and 320 m x 160 m large.

Even though the domain in this demo is quite simple, the concept applies to more complex, realistic scenarios.

The farm layout optimisation is initialized with a regular layout of 32 turbines:

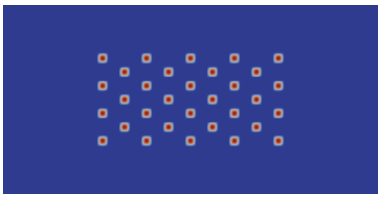


With this configuration, the flow speed with streamlines is:

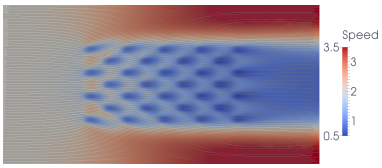



The power extraction by the farm (without taking losses into account) is 46 MW. This is 1.4 MW per turbine (32 turbines) which is unsatisfactory considering that placing a single turbine extracts 2.9 MW.

We can also try a staggered layout:

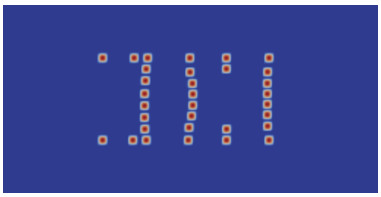


With this configuration, the flow speed with streamlines is:

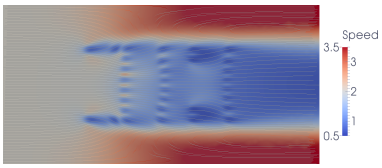



The power extraction by the farm (without taking losses into account) is 64 MW or 2.0 MW per turbine. Thats better but still non-optimal.

Applying the layout optimisation in OpenTidalFarm finishes after 92 iterations. The optimised farm layout is:



The optimization has arranged the turbines to "barrages" perpendicular to the flow. Furthermore, it added small north and east barrages of turbines that force the water to flow through the prependicular "barrages". The associated flow speed with streamlines is:




The power production of the optimised layout is 80 MW, or 2.5 MW per turbine. That is the optimisation increased the power production by 74 % compared to the initial layout!

### Implementation

The first part of the program sets up a steady state shallow water problem, and is nearly identical to the *Sinusoidal wave in a channel* example:

```
from opentidalfarm import *
```

```
# Create a rectangular domain.
domain = FileDomain("mesh/mesh.xml")

# Specify boundary conditions.
bcs = BoundaryConditionSet()
bcs.add_bc("u", Constant((2, 0)), facet_id=1)
bcs.add_bc("eta", Constant(0), facet_id=2)
# The free-slip boundary conditions.
bcs.add_bc("u", Constant((0, 0)), facet_id=3, bctype="weak_dirichlet")

# Set the shallow water parameters
prob_params = SteadySWProblem.default_parameters()
prob_params.domain = domain
prob_params.bcs = bcs
prob_params.viscosity = Constant(2)
prob_params.depth = Constant(50)
prob_params.friction = Constant(0.0025)
```

The next step is to create the turbine farm. In this case, the farm consists of 32 turbines, initially deployed in a regular grid layout. This layout will be the starting guess for the optimization.

```
# Before adding turbines we must specify the type of turbines used in the array.
# Here we used the default BumpTurbine which defaults to being controlled by
# just position. The diameter and friction are set. The minimum distance between
# turbines if not specified is set to 1.5*diameter.
turbine = BumpTurbine(diameter=20.0, friction=12.0)

# A rectangular farm is defined using the domain and the site dimensions.
farm = RectangularFarm(domain, site_x_start=160, site_x_end=480,
                       site_y_start=80, site_y_end=240, turbine=turbine)

# Turbines are then added to the site in a regular grid layout.
farm.add_regular_turbine_layout(num_x=8, num_y=4)

prob_params.tidal_farm = farm
```

Now we can create the shallow water problem

```
problem = SteadySWProblem(prob_params)
```

Next we create a shallow water solver. Here we choose to solve the shallow water equations in its fully coupled form. We also set the dump period to 1 in order to save the results of each optimisation iteration to disk.

```
sol_params = CoupledSWSolver.default_parameters()
sol_params.dump_period = 1
solver = CoupledSWSolver(problem, sol_params)
```

Next we create a reduced functional, that is the functional considered as a pure function of the control by implicitly solving the shallow water PDE. For that we need to specify the objective functional (the value that we want to optimize), the control (the variables that we want to change), and our shallow water solver.

```
functional = PowerFunctional(problem)
control = TurbineFarmControl(farm)
rf_params = ReducedFunctional.default_parameters()
rf_params.automatic_scaling = 5
rf = ReducedFunctional(functional, control, solver, rf_params)
```

As always, we can print all options of the `ReducedFunctional` with:

```
print rf_params
```

Now we can define the constraints for the controls and start the optimisation.

```
lb, ub = farm.site_boundary_constraints()
f_opt = maximize(rf, bounds=[lb, ub], method="L-BFGS-B", options={'maxiter': 100})
```

The example code can be found in `examples/channel-optimization/` in the `OpenTidalFarm` source tree, and executed as follows:

```
$ python channel-optimization.py
```

You can speed up the calculation by using multiple cores (in this case 4) with:

```
$ mpirun -n 4 python channel-optimization.py
```

During the optimization run, OpenTidalFarm creates multiple files for inspection:

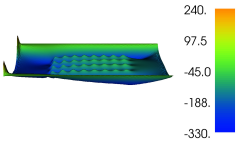
- turbines.pvd: Stores the position and friction values of the turbines at each optimisation iteration.
- iter_*: For each optimisation iteration X, the associated velocity and pressure solutions are stored in a directory named iter_X.
- iterate.dat: A testfile that dumps the optimisation progress, e.g. number of iterations, function value, gradient norm, etc

The pvd files can be opened with the open-source software Paraview.

### 1.3.4 Sensitivity Analysis

#### Introduction

Gradient information may also be used to analyse the sensitivity of the chosen functional to various model parameters - for example the bottom friction:




This enables the designer to identify which parameters may have a high impact upon the quality of the chosen design (as judged by the choice of functional).

#### Implementation

As with other examples, we begin by defining a steady state shallow water problem, once more this is similar to the *Sinusoidal wave in a channel* example except that we define steady flow driven by a 0.1 m head difference across the domain. Note that this is facilitated by defining a strong dirichlet boundary condiditon on the walls:

```
from opentidalfarm import *

# Create a rectangular domain.
domain = FileDomain("mesh/mesh.xml")

# Specify boundary conditions.
bcs = BoundaryConditionSet()
```

```
bcs.add_bc("eta", Constant(0.1), facet_id=1)
bcs.add_bc("eta", Constant(0), facet_id=2)
# The no-slip boundary conditions.
bcs.add_bc("u", Constant((0, 0)), facet_id=3, bctype="strong_dirichlet")
```

Set the shallow water parameters, since we want to extract the spatial variation of the sensitivity of our model parameters, rather than simply defining constant values, we define fields over the domain (in this case we're simply defining a field of a constant value, using dolfin's *interpolate*.

```
prob_params = SteadySWProblem.default_parameters()
prob_params.domain = domain
prob_params.bcs = bcs
V = FunctionSpace(domain.mesh, "CG", 1)
prob_params.viscosity = interpolate(Constant(3), V)
prob_params.depth = interpolate(Constant(50), V)
prob_params.friction = interpolate(Constant(0.0025), V)
```

The next step is to specify the array design for which we wish to analyse the sensitivity. For simplicity we will use the starting guess from the *Farm layout optimization* example; 32 turbines in a regular grid layout. As before we'll use the default turbine type and define the diameter and friction. In practice, one is likely to want to analyse the sensitivity of the optimised array layout - so one would substitute this grid layout with the optimised one.

```
turbine = BumpTurbine(diameter=20.0, friction=12.0)
farm = RectangularFarm(domain, site_x_start=160, site_x_end=480,
                       site_y_start=80, site_y_end=240, turbine=turbine)
farm.add_regular_turbine_layout(num_x=8, num_y=4)
prob_params.tidal_farm = farm
```

Now we can create the shallow water problem

```
problem = SteadySWProblem(prob_params)
```

Next we create a shallow water solver. Here we choose to solve the shallow water equations in its fully coupled form:

```
sol_params = CoupledSWSolver.default_parameters()
sol_params.dump_period = -1
solver = CoupledSWSolver(problem, sol_params)
```

We wish to study the effect that various model parameters have on the power. Thus, we select the `PowerFunctional`

```
functional = PowerFunctional(problem)
```

First let's compute the sensitivity of the power with respect to the turbine positions. So we set the "control" variable to the turbine positions by using `TurbineFarmControl`. We then intialise the `ReducedFunctional`

```
control = TurbineFarmControl(farm)
rf_params = ReducedFunctional.default_parameters()
rf = ReducedFunctional(functional, control, solver, rf_params)
m0 = rf.solver.problem.parameters.tidal_farm.control_array
j = rf.evaluate(m0)
turbine_location_sensitivity = rf.derivative(m0)

print "j for turbine positions: ", j
print "dj w.r.t. turbine positions: ", turbine_location_sensitivity
```

Next we compute the sensitivity of the power with respect to bottom friction. We redefine the control variable using the class `Control` into which we pass the parameter of interest

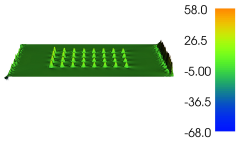
```
control = Control(prob_params.friction)
```

Turbine positions are stored in different data structures (numpy arrays) than functions such as bottom friction (dolfin functions), so we need to use a different reduced functional; the `FenicsReducedFunctional`

```
rf = FenicsReducedFunctional(functional, control, solver)
j = rf.evaluate()
dj = rf.derivative(project=True)
plot(dj, interactive=True, title="Sensitivity with respect to friction")
```

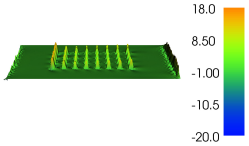
Now compute the sensitivity with respect to depth

```
control = Control(prob_params.depth)
rf = FenicsReducedFunctional(functional, control, solver)
j = rf.evaluate()
dj = rf.derivative(project=True)
print "j with depth = 50 m: ", j
plot(dj, interactive=True, title="Sensitivity with respect to depth at 50m")
```



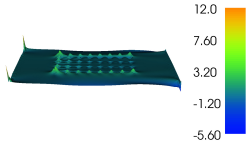

Let's reduce the depth and reevaluate derivative

```
prob_params.depth.assign(Constant(10))
j = rf.evaluate()
dj = rf.derivative(project=True)
print "j with depth = 10 m: ", j
plot(dj, interactive=True, title="Sensitivity with respect to depth at 10m")
```




Finally let's compute the sensitivity with respect to viscosity

```
control = Control(prob_params.viscosity)
rf = FenicsReducedFunctional(functional, control, solver)
j = rf.evaluate()
dj = rf.derivative(project=True)
plot(dj, interactive=True, title="Sensitivity with respect to viscosity")
```




The example code can be found in `examples/channel-sensitivity/` in the `OpenTidalFarm` source tree, and executed as follows:

```
$ python channel-sensitivity.py
```

## 1.4 Programmers reference

The OpenTidalFarm project

### 1.4.1 Domains

This module contains classes to store domain information in OpenTidalFarm. In particular each `Domain` includes:

- the computational mesh;
- subdomains markers (used to identify turbine site areas);
- boundary markers (used to identify where boundary conditions shall be applied).

The mesh, and subdomain and surface makers can be visualised with

```
plot(domain.mesh, title="Mesh")
plot(domain.facet_ids, title="Area ids")
plot(domain.cell_ds, title="Boundary ids")
interactive()
```

#### Create a rectangular domain

**class** opentidalfarm.domains.rectangle_domain.**RectangularDomain**(*x0*, *y0*, *x1*, *y1*, *nx*, *ny*)

Create a rectangular domain.

**Parameters**

- **x0** (*float*) – The x coordinate of the bottom-left.
- **y0** (*float*) – The y coordinate of the bottom-left.
- **x1** (*float*) – The x coordinate of the top-right corner.
- **y1** (*float*) – The y coordinate of the top-right corner.

**cell_ids = None**
A `dolfin.CellFunction` containing the area markers.

**facet_ids = None**
A `dolfin.FacetFunction` containing the surface markers.

**mesh = None**
A `dolfin.Mesh` containing the mesh.

#### Load domain from a mesh file

**class** opentidalfarm.domains.file_domain.**FileDomain**(*mesh_file*, *facet_ids_file=None*, *cell_ids_file=None*)

Create a domain from DOLFIN mesh files (.xml).

**Parameters**

- **mesh_file** (*str*) – The .xml file of the mesh.
- **facet_ids_file** (*str*) – The .xml file containing the facet ids of the mesh. If None, the default is to *mesh_file* + "_facet_region.xml".

- **cell_ids_file** (*str*) – The .xml file containing the cell ids of the mesh. If None, the default is to *mesh_file* + "_physical_region.xml".

**cell_ids = None**
A `dolfin.CellFunction` containing the area markers.

**facet_ids = None**
A `dolfin.FacetFunction` containing the surface markers.

**mesh = None**
A `dolfin.Mesh` containing the mesh.

### 1.4.2 Problems

Problems for OpenTidalFarm

#### Steady shallow water

**class** `opentidalfarm.problems.steady_sw.`**SteadySWProblem**(*parameters*)
Create a steady-state shallow water problem:

$$-\nabla \cdot \nu \nabla u + u \cdot \nabla u + g \nabla \eta + \frac{c_b + c_t}{H} \|u\| u = f_u,$$
$$\nabla \cdot (H u) = 0,$$

where

- $u$ is the velocity,
- $\eta$ is the free-surface displacement,
- $H = \eta + h$ is the total water depth where $h$ is the water depth at rest,
- $f_u$ is the velocity forcing term,
- $c_b$ is the (quadratic) natural bottom friction coefficient,
- $c_t$ is the (quadratic) friction coefficient due to the turbine farm,
- $\nu$ is the viscosity coefficient,
- $g$ is the gravitational constant,

**Parameters** **parameters** – A `SteadySWProblemParameters` object containing the parameters of the problem.

**static default_parameters**()
Returns a `SteadySWProblemParameters` with default parameters.

**class** `opentidalfarm.problems.steady_sw.`**SteadySWProblemParameters**
A parameters set for a `SteadySWProblem`.

Domain parameters:

**Variables** **domain** – The computational domain, see *Domains*.

Physical parameters:

**Variables**

- **depth** – Water depth. Default: 50.0

- **g** – Gravitational constant. Default: 9.81
- **viscosity** – Water viscosity. Default: 3.0
- **friction** – Natural bottom friction. Default: 0.0025
- **rho** – Density of water. Default: 1000.0

Equation parameters:

**Variables**

- **include_advection** – Boolean indicating if the advection is included. Default: True
- **include_viscosity** – Boolean indicating if the viscosity is included. Default: True
- **linear_divergence** – Boolean indicating if the divergence equation is linearised. Default: False
- **f_u** – A source term for the velocity component as a `dolfin.Expression`. Default: Constant((0, 0))

Boundary conditions:

**Variables**

- **bctype** – Specifies how the boundary conditions should be enforced. Valid options are: 'weak_dirichlet', 'strong_dirichlet' or 'flather'. Default: 'strong_dirichlet'
- **bcs** – A `opentidalfarm.boundary_conditions.BoundaryConditionSet` containing a list of boundary conditions for the problem.

Discretization settings:

**Variables** **finite_element** – The finite-element pair to use. Default:

`opentidalfarm.finite_elements.p2p1`

### Multi steady shallow water

**class** `opentidalfarm.problems.multi_steady_sw.`**`MultiSteadySWProblem`**(*parameters*)

Create a shallow water problem consisting of a sequence of (independent) steady-state shallow water problems. More specifically, it solves for each time-level $n$:

$$-\nabla \cdot \nu \nabla u^n + u^n \cdot \nabla u^n + g \nabla \eta^n + \frac{c_b + c_t}{H^n} \|u^n\| u^n = f_u^n,$$
$$\nabla \cdot (H^n u^n) = 0,$$

where

- $u$ is the velocity,
- $\eta$ is the free-surface displacement,
- $H = \eta + h$ is the total water depth where $h$ is the water depth at rest,
- $f_u$ is the velocity forcing term,
- $c_b$ is the (quadratic) natural bottom friction coefficient,
- $c_t$ is the (quadratic) friction coefficient due to the turbine farm,
- $\nu$ is the viscosity coefficient,
- $g$ is the gravitational constant,

**parameter parameters** A `MultiSteadySWProblemParameters` object containing the parameters of the problem.

**static default_parameters()**
Returns a dictionary with the default parameters

**class** opentidalfarm.problems.multi_steady_sw.**MultiSteadySWProblemParameters**
A set of parameters for a `MultiSteadySWProblem`.

The parameters are as described in `opentidalfarm.problems.steady_sw.SteadySWProblemParameters`.

In addition following parameters are available:

Temporal parameters:

**Variables**

- **dt** – The timestep. Default: 1.0.
- **start_time** – The start time. Default: 0.0.
- **finish_time** – The finish time. Default: 100.0.

### Time-dependent shallow water

**class** opentidalfarm.problems.sw.**SWProblem**(*parameters*)
Create a transient shallow water problem:

$$\frac{\partial u}{\partial t} - \nabla \cdot \nu \nabla u + u \cdot \nabla u + g \nabla \eta + \frac{c_b + c_t}{H} \|u\| u = f_u,$$
$$\frac{\partial \eta}{\partial t} + \nabla \cdot (H u) = 0,$$

where

- $u$ is the velocity,
- $\eta$ is the free-surface displacement,
- $H = \eta + h$ is the total water depth where $h$ is the water depth at rest,
- $f_u$ is the velocity forcing term,
- $c_b$ is the (quadratic) natural bottom friction coefficient,
- $c_t$ is the (quadratic) friction coefficient due to the turbine farm,
- $\nu$ is the viscosity coefficient,
- $g$ is the gravitational constant,

**Parameters** **parameters** – A `SWProblemParameters` object containing the parameters of the problem.

**static default_parameters()**
Returns a dictionary with the default parameters

**class** opentidalfarm.problems.sw.**SWProblemParameters**
A set of parameters for a `SWProblem`.

The parameters are as described in `opentidalfarm.problems.steady_sw.SteadySWProblemParameters`.

In addition following parameters are available:

Time parameters:

**Variables**

- **theta** – The theta value for the timestepping-scheme. Default 1.0.
- **dt** – The timestep. Default: 1.0.
- **start_time** – The start time. Default: 0.0.
- **finish_time** – The finish time. Default: 100.0.

Functional time integration paramters (FIXME: Move to reduced functional):

**Variables** **functional_final_time_only** – Boolean indicating if the functional should be integrated over time or evaluated at the end of time only. Default: True.

## Common options

### Boundary conditions

**class** opentidalfarm.boundary_conditions.**BoundaryConditionSet**

Stores a list of boundary conditions. Expression with an attribute named t will be automatically updated to the current timestep during the simultion.

**add_bc**(*function_name*, *expression*, *facet_id*, *bctype='strong_dirichlet'*)

Valid choices for bctype: "weak_dirichlet", "strong_dirichlet", "flather"

**filter**(*function_name=None*, *bctype=None*)

Return a list of boundary conditions that satisfy the given criteria.

**update_time**(*t*, *only_type=None*, *exclude_type=None*)

Update the time attribute for all boundary conditions

### Tidal forcing and bathymetry

**class** opentidalfarm.tidal.**BathymetryDepthExpression**(*filename*, *utm_zone*, *utm_band*, *maxval=10*, *domain=None*)

Bases: Expression

Create a bathymetry depth Expression from a lat/lon NetCDF file, where the depth values stored as "z" field.

**eval**(*values*, *x*)

Evaluates the bathymetry at a point.

**class** opentidalfarm.tidal.**TidalForcing**(*grid_file_name*, *data_file_name*, *ranges*, *utm_zone*, *utm_band*, *initial_time*, *constituents*)

Bases: Expression

Create a TidalForcing Expression from OTPSnc NetCDF files, where the grid is stored in a separate file (with "lon_z", "lat_z" and "mz" fields). The actual data is read from a seperate file with hRe and hIm fields.

**eval**(*values*, *X*)

Evaluates the tidal forcing.

## Advanced options

### Finite element pairs

opentidalfarm.finite_elements.**bdfmp1dg**(*mesh*)

Return a function space U*H on mesh from the BFDM1 space.

opentidalfarm.finite_elements.**bdmp0**(*mesh*)
Return a function space U*H on mesh from the BFDM1 space.

opentidalfarm.finite_elements.**bdmp1dg**(*mesh*)
Return a function space U*H on mesh from the BFDM1 space.

opentidalfarm.finite_elements.**mini**(*mesh*)
Return a function space U*H on mesh from the mini space.

opentidalfarm.finite_elements.**p0p1**(*mesh*)
Return a function space U*H on mesh from the P0P1 space.

opentidalfarm.finite_elements.**p1dgp2**(*mesh*)
Return a function space U*H on mesh from the P1dgP2 space.

opentidalfarm.finite_elements.**p2p1**(*mesh*)
Return a function space U*H on mesh from the P2P1 space.

opentidalfarm.finite_elements.**rt0**(*mesh*)
Return a function space U*H on mesh from the rt0 space.

### 1.4.3 Solvers

A set of solvers for OpenTidalFarm

#### Coupled shallow water solver

**class** opentidalfarm.solvers.coupled_sw_solver.**CoupledSWSolver**(*problem*, *solver_params*)
The coupled solver solves the shallow water equations as a fully coupled system with a $\theta$-timestepping discretization.

For a opentidalfarm.problems.steady_sw.SteadySWProblem, it solves $u$ and $\eta$ such that:

$$u \cdot \nabla u - \nu \Delta u + g \nabla \eta + \frac{c_b + c_t}{H} \|u\| u = f_u,$$
$$\nabla \cdot (H u) = 0.$$

For a opentidalfarm.problems.multi_steady_sw.MultiSteadySWProblem, it solves $u^{n+1}$ and $\eta^{n+1}$ for each timelevel (including the initial time) such that:

$$u^{n+1} \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + g \nabla \eta^{n+1} + \frac{c_b + c_t}{H^{n+1}} \|u^{n+1}\| u^{n+1} = f_u^{n+1},$$
$$\nabla \cdot \left(H^{n+1} u^{n+1}\right) = 0.$$

For a opentidalfarm.problems.sw.SWProblem, and given an initial condition $u^0$ and $\eta^0$ it solves $u^{n+1}$ and $\eta^{n+1}$ for each timelevel such that:

$$\frac{1}{\Delta t}\left(u^{n+1} - u^n\right) + u^{n+\theta} \cdot \nabla u^{n+\theta} - \nu \Delta u^{n+\theta} + g \nabla \eta^{n+\theta} + \frac{c_b + c_t}{H^{n+\theta}} \|u^{n+\theta}\| u^{n+\theta} = f_u^{n+\theta},$$
$$\frac{1}{\Delta t}\left(\eta^{n+1} - \eta^n\right) + \nabla \cdot \left(H^{n+\theta} u^{n+\theta}\right) = 0.$$

with $\theta \in [0, 1]$ and $u^{n+\theta} := \theta u^{n+1} + (1-\theta) u^n$ and $\eta^{n+\theta} := \theta \eta^{n+1} + (1-\theta)\eta^n$.

Furthermore:

- $u$ is the velocity,

- $\eta$ is the free-surface displacement,
- $H = \eta + h$ is the total water depth where $h$ is the water depth at rest,
- $f_u$ is the velocity forcing term,
- $c_b$ is the (quadratic) natural bottom friction coefficient,
- $c_t$ is the (quadratic) friction coefficient due to the turbine farm,
- $\nu$ is the viscosity coefficient,
- $g$ is the gravitational constant,
- $\Delta t$ is the time step.

Some terms, such as the advection or the diffusion term, may be deactivated in the problem specification.

The resulting equations are solved with Newton-iterations and the linear problems solved as specified in the *dolfin_solver* setting in `CoupledSWSolverParameters`.

**static `default_parameters`()**
Return the default parameters for the `CoupledSWSolver`.

**`solve`**(*annotate=True*)
Returns an iterator for solving the shallow water equations.

**class** `opentidalfarm.solvers.coupled_sw_solver.`**`CoupledSWSolverParameters`**
A set of parameters for a `CoupledSWSolver`.

Following parameters are available:

**Variables**

- **dolfin_solver** – The dictionary with parameters for the dolfin Newton solver. A list of valid entries can be printed with:

  ```
  info(NonlinearVariationalSolver.default_parameters(), True)
  ```

  By default, the MUMPS direct solver is used for the linear system. If not available, the default solver and preconditioner of FEniCS is used.
- **dump_period** – Specifies how often the solution should be dumped to disk. Use a negative value to disable it. Default 1.
- **cache_forward_state** – If True, the shallow water solutions are stored for every timestep and are used as initial guesses for the next solve. If False, the solution of the previous timestep is used as an initial guess. Default: True
- **print_individual_turbine_power** – Print out the turbine power for each turbine. Default: False
- **quadrature_degree** – The quadrature degree for the matrix assembly. Default: -1 (automatic)
- **cpp_flags** – A list of cpp compiler options for the code generation. Default: [”-O3”, “-ffast-math”, “-march=native”]
- **revolve_parameters** – The adjoint checkpointing settings as a set of the form (strategy, snaps_on_disk, snaps_in_ram, verbose). Default: None
- **output_dir** – The base directory in which to store the file ouputs. Default: *os.curdir*
- **output_turbine_power** – Output the power generation of the individual turbines. Default: False

### Pressure-correction shallow water solver

**class** opentidalfarm.solvers.ipcs_sw_solver.**IPCSSWSolver**(*problem*, *parameters*)

This incremental pressure correction scheme (IPCS) is an operator splitting scheme that follows the idea of Goda [1] and Simo [2]. This scheme preserves the exact same stability properties as Navier-Stokes and hence does not introduce additional dissipation in the flow (FIXME: needs verification).

The idea is to replace the unknown free-surface with an approximation. This is chosen as the free-surface solution from the previous solution.

The time discretization is done using a $\theta$-scheme, the convection, friction and divergence are handled semi-implicitly. Thus, we have a discretized version of the shallow water equations as

$$\frac{1}{\Delta t}\left(u^{n+1}-u^n\right) - \nabla\cdot\nu\nabla u^{n+\theta} + u^*\cdot\nabla u^{n+\theta} + g\nabla\eta^{n+\theta} + \frac{c_b+c_t}{H^n}\|u^n\|u^{n+\theta} = f_u^{n+\theta},$$
$$\frac{1}{\Delta t}\left(\eta^{n+1}-\eta^n\right) + \nabla\cdot\left(H^n u^{n+\theta}\right) = 0,$$

where $\square^{n+\theta} = \theta\square^{n+1} + (1-\theta)\square^n, \theta\in[0,1]$ and $u^* = \frac{3}{2}u^n - \frac{1}{2}u^{n-1}$.

This convection term is unconditionally stable, and with $\theta = 0.5$, this equation is second order in time and space [2] (FIXME: Needs verification).

For the operator splitting, we use the free-surface solution from the previous timestep as an estimation, giving an equation for a tentative velocity, $\tilde{u}^{n+1}$:

$$\frac{1}{\Delta t}\left(\tilde{u}^{n+1}-u^n\right) - \nabla\cdot\nu\nabla\tilde{u}^{n+\theta} + u^*\cdot\nabla\tilde{u}^{n+\theta} + g\nabla\eta^{n} + \frac{c_b+c_t}{H^n}\|u^n\|\tilde{u}^{n+\theta} = f_u^{n+\theta}.$$

This tenative velocity does not satisfy the divergence equation, and thus we define a velocity correction $u^c = u^{n+1} - \tilde{u}^{n+1}$. Substracting the second equation from the first, we see that

$$\frac{1}{\Delta t}u^c - \theta\nabla\cdot\nu\nabla u^c + \theta u^*\cdot\nabla u^c + g\theta\nabla\left(\eta^{n+1}-\eta^n\right) + \theta\frac{c_b+c_t}{H^n}\|u^n\|u^c = 0,$$
$$\frac{1}{\Delta t}\left(\eta^{n+1}-\eta^n\right) + \theta\nabla\cdot\left(H^n u^c\right) = -\nabla\cdot\left(H^n\tilde{u}^{n+\theta}\right).$$

The operator splitting is a first order approximation, $O(\Delta t)$, so we can, without reducing the order of the approximation simplify the above to

$$\frac{1}{\Delta t}u^c + g\theta\nabla\left(\eta^{n+1}-\eta^n\right) = 0,$$
$$\frac{1}{\Delta t}\left(\eta^{n+1}-\eta^n\right) + \theta\nabla\cdot\left(H^n u^c\right) = -\nabla\cdot\left(H^n\tilde{u}^{n+\theta}\right),$$

which is reducible to the problem:

$$\eta^{n+1} - \eta^n - g\Delta t^2\theta^2\nabla\cdot\left(H^{n+1}\nabla\left(\eta^{n+1}-\eta^n\right)\right) = -\Delta t\nabla\cdot\left(H^n\tilde{u}^{n+\theta}\right).$$

The corrected velocity is then easily calculated from

$$u^{n+1} = \tilde{u}^{n+1} - \Delta t g\theta\nabla\left(\eta^{n+1}-\eta^n\right)$$

**The scheme can be summarized in the following steps:**

1. Replace the pressure with a known approximation and solve for a tenative velocity $\tilde{u}^{n+1}$.

[1] Goda, Katuhiko. *A multistep technique with implicit difference schemes for calculating two-or three-dimensional cavity flows.* Journal of Computational Physics 30.1 (1979): 76-95.

[2] Simo, J. C., and F. Armero. *Unconditional stability and long-term behavior of transient algorithms for the incompressible Navier-Stokes and Euler equations.* Computer Methods in Applied Mechanics and Engineering 111.1 (1994): 111-154.

2. Solve a free-surface correction equation for the free-surface, $\eta^{n+1}$
3. Use the corrected pressure to find the velocity correction and calculate $u^{n+1}$
4. Update t, and repeat.

Remarks:

•This solver only works with transient problems, that is with `opentidalfarm.problems.sw.SWProblem`.

•This solver supports large eddy simulation (LES). The LES model is implemented via the `opentidalfarm.solvers.les.LES` class.

**solve**(*annotate=True*)
Solve the shallow water equations

**class** opentidalfarm.solvers.ipcs_sw_solver.**IPCSSWSolverParameters**
A set of parameters for a `IPCSSWSolver`.

Performance parameters:

**Variables**

- **dolfin_solver** – The dictionary with parameters for the dolfin Newton solver. A list of valid entries can be printed with:

  ```
  info(NonlinearVariationalSolver.default_parameters(), True)
  ```

  By default, the MUMPS direct solver is used for the linear system. If not availabe, the default solver and preconditioner of FEniCS is used.
- **quadrature_degree** – The quadrature degree for the matrix assembly. Default: -1 (auto)
- **cpp_flags** – A list of cpp compiler options for the code generation. Default: [”-O3”, “-ffast-math”, “-march=native”]

Large eddy simulation parameters:

**Variables**

- **les_model** – De-/Activates the LES model. Default: True
- **les_model_parameters** – A dictionary with parameters for the LES model. Default: {‘smagorinsky_coefficient’: 1e-2}

### 1.4.4 Functionals

Initialises the various functionals. All functionals should overload the Prototype, as this allows them to be combined (added, subtracted and scaled). Functionals should be grouped by type; Power / Cost / Environment etc.

#### Prototype Functional

Functional objects are overloads of the prototype functional. This allows functionals to be combined and scaled. The initialisation and ‘Jt’ methods should be overloaded.

### Power Functionals

**class** opentidalfarm.functionals.power_functionals.**PowerCurveFunctional**(*farm*)
TODO: doesn't work yet...

**class** opentidalfarm.functionals.power_functionals.**PowerFunctional**(*problem*)
Implements a simple functional of the form:

$$J(u, m) = \int \rho c_t ||u||^3 \, dx,$$

where $c_t$ defines the friction field due to the turbines.

**Parameters** **problem** (*Instance of the problem class.*) – The problem for which the functional is being computed.

**Jt**(*state*, *turbine_field*)
Computes the power output of the farm.

**Parameters**

- **state** (*UFL*) – Current solution state
- **turbine_field** (*UFL*) – Turbine friction field

**Jt_individual**(*state*, *i*)
Computes the power output of the i'th turbine.

**Parameters**

- **state** (*UFL*) – Current solution state
- **i** (*Integer*) – refers to the i'th turbine

**force**(*state*, *turbine_field*)
Computes the force field over turbine field

**Parameters**

- **state** (*UFL*) – Current solution state.
- **turbine_field** (*UFL*) – Turbine friction field

**force_individual**(*state*, *i*)
Computes the total force on the i'th turbine

**Parameters**

- **state** (*UFL*) – Current solution state
- **i** (*Integer*) – refers to the i'th turbine

**power**(*state*, *turbine_field*)
Computes the power field over the domain.

**Parameters**

- **state** (*UFL*) – Current solution state.
- **turbine_field** (*UFL*) – Turbine friction field

### 1.4.5 Turbine farms

**class** opentidalfarm.farm.**Farm**(*domain*, *turbine=None*, *site_ids=None*)
Bases: opentidalfarm.farm.base_farm.BaseFarm

Extends BaseFarm. Creates a farm from a mesh.

This class holds the turbines within a site defined by a mesh with a turbine site marked by *1*.

**class** opentidalfarm.farm.**ContinuumFarm**(*domain*, *site_ids=(0, )*, *friction_function=None*)
Bases: object

Creates a farm from a mesh using the continues turbine representation.

**friction_function**

**update**()

**class** opentidalfarm.farm.**RectangularFarm**(*domain*, *site_x_start*, *site_x_end*, *site_y_start*, *site_y_end*, *turbine=None*, *site_ids=None*)
Bases: opentidalfarm.farm.farm.Farm

Extends Farm. Defines a rectangular Farm.

This class holds the turbines within a rectangular site.

**add_regular_turbine_layout**(*num_x*, *num_y*, *x_start=None*, *x_end=None*, *y_start=None*, *y_end=None*)
Adds a rectangular turbine layout to the farm.

A rectangular turbine layout with turbines evenly spread out in each direction across the given rectangular site.

**Parameters**

- **turbine** (*Turbine object.*) – Defines the type of turbine to add to the farm.
- **num_x** (*int*) – The number of turbines placed in the x-direction.
- **num_y** (*int*) – The number of turbines placed in the y-direction.
- **x_start** (*float*) – The minimum x-coordinate of the site.
- **x_end** (*float*) – The maximum x-coordinate of the site.
- **y_start** (*float*) – The minimum y-coordinate of the site.
- **y_end** (*float*) – The maximum y-coordinate of the site.

**Raises** ValueError

**add_staggered_turbine_layout**(*num_x*, *num_y*, *x_start=None*, *x_end=None*, *y_start=None*, *y_end=None*)
Adds a rectangular, staggered turbine layout to the farm.

A rectangular turbine layout with turbines evenly spread out in each direction across the given rectangular site.

**Parameters**

- **turbine** (*Turbine object.*) – Defines the type of turbine to add to the farm.
- **num_x** (*int*) – The number of turbines placed in the x-direction.
- **num_y** (*int*) – The number of turbines placed in the y-direction (will be one less on every second row).
- **x_start** (*float*) – The minimum x-coordinate of the site.

- **x_end** (*float*) – The maximum x-coordinate of the site.
- **y_start** (*float*) – The minimum y-coordinate of the site.
- **y_end** (*float*) – The maximum y-coordinate of the site.

**Raises** ValueError

**site_boundary_constraints**()
Returns the site boundary constraints for a rectangular site.

These constraints ensure that the turbine positions remain within the turbine site during optimisation.

**Raises** ValueError

**Returns** Tuple of lists of length equal to the twice the number of turbines. Each list contains dolfin_adjoint.Constant objects of the upper and lower bound coordinates.

**site_x_end**
The maximum x-coordinate of the site.

**Getter** Returns the maximum x-coordinate of the site.

**Type** float

**site_x_start**
The minimum x-coordinate of the site.

**Getter** Returns the minimum x-coordinate of the site.

**Type** float

**site_y_end**
The maximum y-coordinate of the site.

**Getter** Returns the maximum y-coordinate of the site.

**Type** float

**site_y_start**
The minimum y-coordinate of the site.

**Getter** Returns the minimum y-coordinate of the site.

**Type** float

### Create a rectangular farm

**class** opentidalfarm.farm.rectangular_farm.**RectangularFarm**(*domain*, *site_x_start*, *site_x_end*, *site_y_start*, *site_y_end*, *turbine=None*, *site_ids=None*)

Extends Farm. Defines a rectangular Farm.

This class holds the turbines within a rectangular site.

**add_regular_turbine_layout**(*num_x*, *num_y*, *x_start=None*, *x_end=None*, *y_start=None*, *y_end=None*)
Adds a rectangular turbine layout to the farm.

A rectangular turbine layout with turbines evenly spread out in each direction across the given rectangular site.

**Parameters**

- **turbine** (*Turbine object.*) – Defines the type of turbine to add to the farm.

- **num_x** (*int*) – The number of turbines placed in the x-direction.
- **num_y** (*int*) – The number of turbines placed in the y-direction.
- **x_start** (*float*) – The minimum x-coordinate of the site.
- **x_end** (*float*) – The maximum x-coordinate of the site.
- **y_start** (*float*) – The minimum y-coordinate of the site.
- **y_end** (*float*) – The maximum y-coordinate of the site.

**Raises** ValueError

**add_staggered_turbine_layout**(*num_x*, *num_y*, *x_start=None*, *x_end=None*, *y_start=None*, *y_end=None*)

Adds a rectangular, staggered turbine layout to the farm.

A rectangular turbine layout with turbines evenly spread out in each direction across the given rectangular site.

**Parameters**

- **turbine** (*Turbine object.*) – Defines the type of turbine to add to the farm.
- **num_x** (*int*) – The number of turbines placed in the x-direction.
- **num_y** (*int*) – The number of turbines placed in the y-direction (will be one less on every second row).
- **x_start** (*float*) – The minimum x-coordinate of the site.
- **x_end** (*float*) – The maximum x-coordinate of the site.
- **y_start** (*float*) – The minimum y-coordinate of the site.
- **y_end** (*float*) – The maximum y-coordinate of the site.

**Raises** ValueError

**site_boundary_constraints**()

Returns the site boundary constraints for a rectangular site.

These constraints ensure that the turbine positions remain within the turbine site during optimisation.

**Raises** ValueError

**Returns** Tuple of lists of length equal to the twice the number of turbines. Each list contains dolfin_adjoint.Constant objects of the upper and lower bound coordinates.

**site_x_end**

The maximum x-coordinate of the site.

**Getter** Returns the maximum x-coordinate of the site.

**Type** float

**site_x_start**

The minimum x-coordinate of the site.

**Getter** Returns the minimum x-coordinate of the site.

**Type** float

**site_y_end**

The maximum y-coordinate of the site.

**Getter** Returns the maximum y-coordinate of the site.

**Type** float

**site_y_start**
The minimum y-coordinate of the site.

**Getter** Returns the minimum y-coordinate of the site.

**Type** float

### Create a base farm

**class** opentidalfarm.farm.base_farm.**BaseFarm**(*domain=None*, *turbine=None*, *site_ids=None*)
A base Farm class from which other Farm classes should be derived.

**add_turbine**(*coordinates*)
Add a turbine to the farm at the given coordinates.

Creates a new turbine of the same specification as the prototype turbine and places it at coordinates.

**Parameters** **coordinates** (list()) – The x-y coordinates where the turbine should be placed.

**control_array**
A serialized representation of the farm based on the controls.

**Returns** A serialized representation of the farm based on the controls.

**Return type** numpy.ndarray

**friction_function**

**minimum_distance_constraints**()
Returns an instance of MinimumDistanceConstraints.

**Returns** An instance of InequalityConstraint defining the minimum distance between turbines.

**Return type** opentidalfarm.farm.MinimumDistanceConstraints

**number_of_turbines**
The number of turbines in the farm. :returns: The number of turbines in the farm. :rtype: int

**set_turbine_positions**(*positions*)
Sets the turbine position and an equal friction parameter.

**Parameters** **positions** (*list*) – List of tuples containint x-y coordinates of turbines to be added.

**site_boundary_constraints**()
Raises NotImplementedError if called.

**turbine_frictions**
The friction coefficients of turbines within the farm. :returns: The friction coefficients of turbines within the farm. :rtype: list()

**turbine_positions**
The positions of turbines within the farm. :returns: The positions of turbines within the farm. :rtype: list()

**turbine_specification**

**update**()

### 1.4.6 Reduced functional

**class** opentidalfarm.reduced_functional.**ReducedFunctional**(*functional*, *controls*, *solver*, *parameters*)

Following parameters are expected:

**Variables**

- **functional** – a PrototypeFunctional class.
- **controls** – a TurbineFarmControl or dolfin_adjoint.DolfinAdjointControl class.
- **solver** – a Solver object.
- **parameters** – a ReducedFunctionalParameters object.

This class has a parameter attribute for further adjustments.

**static default_parameters**()
Return the default parameters for the ReducedFunctional.

**derivative**(*m_array*, *forget=True*, ***kwargs*)
Computes the first derivative of the functional with respect to its parameters by solving the adjoint equations.

**evaluate**(*m*, *annotate=True*)
Return the functional value for the given parameter array.

**class** opentidalfarm.reduced_functional.**ReducedFunctionalParameters**
A set of parameters for a ReducedFunctional.

Following parameters are available:

**Variables**

- **scale** – A scaling factor. Default: 1.0
- **automatic_scaling** – The reduced functional will be automatically scaled such that the maximum absolute value of the initial gradient is equal to the specified factor. Set to False to deactivate the automatic scaling. Default: 5.
- **load_checkpoints** – If True, the checkpoints are loaded from file and used. Default: False
- **save_checkpoints** – Automatically store checkpoints after each optimisation iteration. Default: False
- **checkpoint_basefilename** – The base filename (without extensions) for storing or loading the checkpoints. Default: 'checkpoints'.

**class** opentidalfarm.reduced_functional.**TurbineFarmControl**(*farm*)
This class is required to that the parameter set works with dolfin-adjoint.

### 1.4.7 Optimization

#### Site constraints

#### Minimum distance constraints

**class** opentidalfarm.optimisation_helpers.**ConvexPolygonSiteConstraint**(*farm*, *vertices*)

Bases: InequalityConstraint

Generates the inequality constraints for generic polygon constraints. The parameter polygon must be a list of point coordinates that describes the site edges in anti-clockwise order.

**function**(*m*)

**jacobian**(*m*)

**length**()

**output_workspace**()

**class** opentidalfarm.optimisation_helpers.**DomainRestrictionConstraints**(*config*, *feasible_area*, *attraction_center*)

Bases: InequalityConstraint

**function**(*m*)

**jacobian**(*m*)

**length**()

opentidalfarm.optimisation_helpers.**friction_constraints**(*config*, *lb=0.0*, *ub=None*)
This function returns the constraints to ensure that the turbine friction controls remain reasonable.

opentidalfarm.optimisation_helpers.**get_distance_function**(*config*, *domains*)

opentidalfarm.optimisation_helpers.**get_domain_constraints**(*config*, *feasible_area*, *attraction_center*)

opentidalfarm.optimisation_helpers.**position_constraints**(*config*)
This function returns the constraints to ensure that the turbine positions remain inside the domain.

### 1.4.8 Internal classes

**class** opentidalfarm.memoize.**MemoizeMutable**(*fn*, *hash_keys=False*)
Implements a memoization function to avoid duplicated functional (derivative) evaluations

**class** opentidalfarm.functionals.time_integrator.**TimeIntegrator**(*problem*, *functional*, *final_only*)

Bases: object

**add**(*time*, *state*, *tf*, *is_final*)

**dolfin_adjoint_functional**()

**integrate**()

**class** opentidalfarm.helpers.**FrozenClass**
Bases: object

A class which can be (un-)frozen. If the class is frozen, no attributes can be added to the class.

**class** opentidalfarm.helpers.**OutputWriter**(*functional*)
Bases: object

Suite of tools to write output to disk

**individual_turbine_power**(*solver*)
Print out the individual turbine's power

**class** opentidalfarm.helpers.**StateWriter**(*solver*, *callback=None*)

**output_files**(*basename*)

**p_output_projector**(*mesh*)

**u_output_projector**(*mesh*)

**write**(*state*)

opentidalfarm.helpers.**cpu0only**(*f*)
A decorator class that only evaluates on the first CPU in a parallel environment.

opentidalfarm.helpers.**function_eval**(*func*, *point*)
A parallel safe evaluation of dolfin functions

opentidalfarm.helpers.**get_rank**()
The processor number.

**Returns** int – The processor number.

opentidalfarm.helpers.**norm_approx**(*u*, *alpha=0.0001*)
A smooth approximation to $\|u\|$:

$$\|u\|_\alpha = \sqrt{(u^2 + alpha^2)}$$

**Parameters**

- **u** – The coefficient.
- **alpha** – The approximation coefficient.

**Returns** ufl expression – the approximate norm of u.

opentidalfarm.helpers.**smooth_uflmin**(*a*, *b*, *alpha=1e-08*)
A smooth approximation to $\min(a, b)$:

$$\min_\alpha(a, b) = a - \frac{1}{2}(\|a - b\|_\alpha + a - b)$$

**Parameters**

- **a** – First argument to $\min$.
- **b** – Second argument to $\min$.
- **alpha** – The approximation coefficient.

**Returns** ufl expression – the approximate $\min$ function.

opentidalfarm.helpers.**test_gradient_array**(*J*, *dJ*, *x*, *seed=0.01*, *perturbation_direction=None*, *number_of_tests=5*, *plot_file=None*)
Checks the correctness of the derivative dJ. x must be an array that specifies at which point in the parameter space the gradient is to be checked. The functions J(x) and dJ(x) must return the functional value and the functional derivative respectivaly.

This function returns the order of convergence of the Taylor series remainder, which should be 2 if the gradient is correct.

**class** opentidalfarm.solvers.les.**LES**(*V*, *u*, *smagorinsky_coefficient*)
Bases: object

A solver for computing the eddy viscosity by solving:

$$e = (sw)^2 I$$

where $e$ is the eddy viscosity, $s$ is the smagorinsky coefficient, $w = \sqrt{\text{cell volume}}$ is the filter width, and $I$ is the second invariant defined as:

$$I = \sum_{1 \leq i,j \leq 2} 2S_{i,j}^2, \quad S = \frac{1}{2}\left(\nabla u + \nabla u^T\right)$$

Parameters:

**Parameters**

- **V** – The function space for the the eddy viscosity.
- **u** – The velocity function.
- **smagorinsky_coefficient** – The smagorinsky coefficient.

**Variables** **eddy_viscosity** – The smagorinsky coefficient.

**solve**()

Update the eddy viscosity solution for the current velocity.

**Returns** The eddy viscosity.

## 1.5 Contributing to OpenTidalFarm

If you wish to contribute to the development of OpenTidalFarm please adhere to the following *guidelines* on Python *coding style* and *language rules*.

It is **strongly** recommended that contributors read through the entirety of the Google Python Style Guide.

Key points are summarised below.

### 1.5.1 Style Guide

#### Formatting

**Line length** should be limited to **80** characters. Use Python's implicit line joining inside parentheses, brackets and braces.

```
string = ("This is a very long string containing an example of how to "
          "implicitly join strings over multiple lines using parentheses.")
```

Similarly for a long if statement:

```
if (we_have_long_variable_names or lots_of_comparisons and
    need_to_break_onto_another_line):
    # We should use Python's implicit line joining within parentheses.
```

It is permissible to use more than 80 characters in a line when including a URL in a comment, for example:

```
# Find more information at:
# http://www.a-ridiculously-long-url-which-spans-more-than-80-characters-is-allowed-in-a-comment.com
```

**Indentation** should be with **4 spaces**. When a line is implicitly continued (as demonstrated in the line length section) the wrapped elements should be aligned vertically or using a hanging indent of 4 spaces.

```
# Aligned with the opening delimiter
foo = long_function_name(variable_one, variable_two, variable_three,
                         variable_four, variable_five)

# Aligned using a hanging indent of 4 spaces; nothing on the first line
foo = long_function_name(
    variable_one, variable_two, variable_three, variable_four, variable_five)
```

**Whitespace** should follow normal typographic rules, i.e. put a space after a comma but not before.

Do not put whitespace:

- inside parentheses, brackets, or braces,
- before a comma, colon or semicolon, and
- before opening parentheses that starts an argument list, indexing or slicing.

A single space should be added around binary operators for:

- assignment (=),
- comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), and
- Booleans (and, or, not).

However, spaces should *not* be added around the assignment operator (=) when used to indicate a keyword argument or a default value. I.e. you should do this:

```
functions_with_default_arguments(argument_one=10.0, argument_two=20.0)
```

Many more examples regarding whitespace may be again found in the whitespace section of the Google Python Style Guide.

**Blank lines** should be added as such:

- Two blank lines between top-level definitions, be they function or class definitions.
- One blank line between method definitions and between the class line and the first method. Use single blank lines as you judge appropriate within functions or methods.

### Naming Convention

The following convention should be used for naming:

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_variable_name, instance_variable_name, function_parameter_name, local_variable_name.

### Imports formatting

Imports should be at the top of the file and should occur on separate lines:

```
import numpy
import dolfin
```

They should also be ordered from most generic to least generic:

- standard library imports (such as math),
- third-party imports (such as opentidalfarm),

- application-specific imports (such as `farm`).

### Commenting and Documentation

Documenting your work is crucial for to allowing other users and developers to quickly understand what your work does and how it works. For example a docstring for a function should give enough information to write a call it without reading the function's code. A docstring should describe the function's calling syntax and its semantics, not its implementation. For tricky code, comments alongside the code are more appropriate than using docstrings.

OpenTidalFarm uses Sphinx documentation thus a certain syntax is required, examples are given below.

For a module:

```
"""
.. module:: example_module
   :synopsis: Brief description of the module.

"""
```

For a class:

```
class ExampleClass(object):
    """A brief description of the class.

    A longer description of the class.

    .. note::

        Any notes you may wish to highlight in the online documentation.

    """
    # Implementation of ExampleClass...
```

And an example for a function:

```
def public_function_with_sphinx_docstring(name, state=None):
    """This function does something.

    :param name: The name to use.
    :type name: str.
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError

    """
    # Implementation of public_function_with_sphinx_docstring...
```

Finally, comments should also be added within the code to explain where it may not be immediately obvious what is being done. These comments should be well written with correct spelling, punctuation and grammar.

## 1.5.2 Language Rules

Most of the information regarding language rules in the Google Python Style Guide is fairly obvious but a few important points are highlighted here.

**List comprehensions** when used correctly can create lists in a very concise manner, however they should not be used in complicated situations as they can become hard to read.

**Properties** may be used to **control access** to class data members. For example a class which defines the turbine farm may be initialized with the coordinates defining the boundary for the site. Once initialized it does not make sense to resize the site (as turbines may no longer lie within its bounds) but the user may wish to still access these values. In Python there is no way to truly make certain data private but the following convention is ususally adopted.

For read-only data the *property* decorator is used:

```
class Circle(object):
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """The radius of the circle."""
        return self._radius
```

Thus the user may still access the radius of the circle without changing it:

```
>>> circle = Circle(10.0)
>>> circle.radius
10.0
>>> circle.radius = 15.0
AttributeError: can't set attribute
```

If the user wishes the provide full access to a data member it can be done so using the built-in property function. This also provides a convenient way to allow a number of properties to be based upon a single property.

```
class Circle(object):
    def __init__(self, radius):
        self._radius = radius

    def _get_radius(self):
        return self._radius

    def _set_radius(self, radius):
        self._radius = radius

    radius = property(_get_radius, _set_radius, "Radius of circle")

    def _get_diameter(self):
        return self._radius*2

    def _set_diameter(self, diameter):
        self._radius = diameter*0.5

    diameter = property(_get_diameter, _set_diameter, "Diameter of circle")
```

Thus we may do the following:

```
>>> circle = Circle(10.0)
>>> circle.diameter
20.0
>>> circle.diameter = 10.0
>>> circle.radius
5.0
```

### 1.5.3 Logging using dolfin.log

It is strongly encouraged that developers make use of the logging capability of `dolfin`. The verbosity of the logger during runtime may be altered by the user allowing for easier debugging.

The logger is included by `dolfin` and has a number of verbosity levels given in the table below.

| Log Level | Value |
|---|---|
| ERROR | 40 |
| WARNING | 30 |
| INFO | 20 |
| PROGRESS | 16 |
| DBG / DEBUG | 10 |

Controlling the verbosity of what the logger displays during runtime is simple:

```
import dolfin
# Can be any of the values from the table above
dolfin.set_log_level(INFO)
```

Using the logger is simple, for example when adding turbines to a farm it may be useful to know how many turbines are being added to the farm (for which we would set the log level to INFO). In certain cases it may useful to know when each turbine is being added, in which case we would use the PROGRESS log level:

```
import dolfin

class RectangularFarm(object):
    # Implementation of RectangularFarm ...


    def add_regular_turbine_layout(self, num_x, num_y):
        """Adds turbines to the farm in a regularly spaced array."""

        dolfin.log(dolfin.INFO, "Adding %i turbines to the farm..."
                   % (num_x*num_y))

        added = 1
        total = num_x*num_y
        for x in num_x:
            for y in num_y:
                dolfin.log(dolfin.PROGRESS, "Adding turbine %i of %i..."
                           % (added, total))
                # ...add turbines to the farm
                added += 1

        dolfin.log(dolfin.INFO, "Added %i turbines to the farm."
                   % (added))
```

More information may be found in the documentation.

It is also suggested that for computationally expensive functions that the `dolfin.Progress` bar is used. An example from the documentation is shown below.

```
>>> import dolfin
>>> dolfin.set_log_level(dolfin.PROGRESS)
>>> n = 10000000
>>> progress_bar = dolfin.Progress("Informative progress message...", n)
>>> for i in range(n):
...     progress_bar += 1
...
```

```
Informative progress message... [>                                    ] 0.0%
Informative progress message... [=>                                   ] 5.2%
Informative progress message... [====>                                ] 11.1%
Informative progress message... [======>                              ] 17.0%
```

### 1.5.4 Adding documented examples

The documentation for examples is automatically generated from the source code using pylit.

Follow these steps to add an example:

1. Create a new subdirectory in `examples/` and add the documented Python source code (use for example existing examples for references).
2. Add the example to the *build_examples* task in `docs/Makefile` (again use existing commands as a template).
3. Add the example into the list in `examples.rst` to add the hyperlink.
4. Run "make html" in `docs/`, check that the documentation looks as expected (open `_build/html/index.html` in an webbrowser).
5. Add the generated rst file in `docs/examples/.../` to the git repository. Commit, and check that the documentation is correct in the readthedocs OpenTidalFarm documentation.

## 1.6 Developers

OpenTidalFarm is developed by

- Simon Funke, Simula Research Laboratory
- Patrick Farrell, Oxford University
- Matthew Piggott, Imperial College London
- Stephan Kramer, Imperial College London
- David Culley, Imperial College London
- George Barnett, Imperial College London

## 1.7 Citing

Please cite the following paper if you are using OpenTidalFarm:

- **SW Funke, PE Farrell, MD Piggott (2014)**. *Tidal turbine array optimisation using the adjoint approach* Renewable Energy, 63, pp. 658-673. doi:10.1016/j.renene.2013.09.031. arXiv:1304.1768 [cs.MS]. [PDF].

For work on global optimisation of turbine locations through the use of a hybrid global-local optimisation schemes and wake models pease see:

- **GL Barnett, SW Funke, MD Piggott (2014)**. *Hybrid global-local optimisation algorithms for the layout design of tidal turbine arrays*, submitted. arXiv:1410.2476 [math.OC] [PDF].

For work on the addition of cost models into the OpenTidalFarm framework please see:

- **DM Culley, SW Funke, SC Kramer, MD Piggott (2014)**. *Integration of cost modelling within the micro-siting design optimisation of tidal turbine arrays*, submitted. doi:10.6084/m9.figshare.1219374. [PDF].

For a general piece on the role OpenTidalFarm might play in the overall tidal turbine array design process see:

- **DM Culley, SW Funke, SC Kramer, MD Piggott (2014)**. *A hierarchy of approaches for the optimal design of tidal turbine farms*, submitted. doi:10.684/m9.figshare.1219375. [PDF].

Finally, OpenTidalFarm relies on external tools to achieve its goals, such as:

- **A Logg, K-A Mardal, GN Wells et al. (2013)**. *Automated Solution of Differential Equations by the Finite Element Method*, Springer [Webpage, PDF].
- **SW Funke and PE Farrell (2013)**. *A framework for automated PDE-constrained optimisation*, submitted. arXiv:1302.3894 [cs.MS] [PDF].
- **PE Farrell, DA Ham, SW Funke and ME Rognes (2013)**. *Automated derivation of the adjoint of high-level transient finite element programs*, SIAM Journal on Scientific Computing 35.4, pp. C369-C393. doi:10.1137/120873558. arXiv:1204.5577 [cs.MS]. [PDF].

### 1.7.1 Posters and presentations

OpenTidalFarm at the International Conference on Ocean Energy, Halifax, Nova Scotia, 2014:

Poster presentation:

- **DM Culley, SW Funke, SC Kramer, MD Piggott (2014)**. *Resource assessment of tidal sites through array optimisation of the number of turbines and the micro-siting design*, doi:10.6084/m9.figshare.1219384. [PDF].

CHAPTER 2

# Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER 3

# Licence

OpenTidalFarm is an open source project that can be freely used under the GNU GPL version 3 licence.

## f



## o

## A



## B



## C



## D



## E



## F

## P



## R



## S



## T



## U

## W