**Mureil Python Software**

# Introduction

## Document Summary

**Author:** Marcelle Gannon – [mgannon@student.unimelb.edu.au](mailto:mgannon@student.unimelb.edu.au), [marcelle.gannon@gmail.com](mailto:marcelle.gannon@gmail.com)

**Purpose:** This document outlines the status of the Python software written for the MUREIL energy system modelling project. It provides an overview of how the code fits together, how to configure simulations and how to extend and update the models, as well as some Python tips. The document should be read in conjunction with the extensive commenting in the code.

**Software:** see code.google.com, project mureil-ga, in the trunk. Written for Python v2.7.3.

**Document history:**

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Summary** |
| 28 February 2013 | Marcelle Gannon | Initial version |
|  |  |  |

**People named in this document:** all from the University of Melbourne – Roger Dargaville, Robert Huva, Elly Hutton, Steven Thomas, Peter Rayner.

## Outline of modelling approach and history

### History

The software is based on work done by Roger and Robert in IDL, for the MUREIL prototype paper. This was converted to Python by Robert and Steven in 2012. Marcelle then wrote the configurable framework and class-based modelling system in Python in late 2012 – early 2013.

### Meteorological data and site selection

The MUREIL model aims to model a renewable energy system on an hour-by-hour or finer timestep. Long timeseries (e.g. 2 years) of meteorological data, over a large number of sites, are used to determine the expected power output from wind, solar and other renewable sources. The operation of the total energy system (renewables plus non-renewable sources) is modelled relative to a system demand timeseries, which may be static or calculated.

Typically, to keep computation manageable, a subset of the available data is used. Robert has developed off-line algorithms for site selection, based on the correlation between the meteorological data timeseries.

### Genetic algorithm

A genetic algorithm is used to find an optimal combination of renewable energy installation sizes and non-renewable installations. The installation sizes are parameterised and the parameter array and a cost function (the main focus of the modelling efforts) given to the genetic algorithm for optimisation. The genetic algorithm is generic, and the Python code was written by Steven with direction from Peter.

### Multi-period operation - Transition

The simulation is written to be suitable for modelling multiple periods, for example decades. This gives multiple opportunities to install new renewable or other generation capacity, and for modelling of changing fossil fuel prices, carbon taxes and generator capital costs. The optimal transition from the currently installed capacity to an energy system in many decades’ time can be found. Each model allows for a startup state to be specified.

### Modularity and backwards compatibility

Modularity is a key feature of this software implementation. The configuration framework is quite generic and can be used to hook up a combination of any master model and compatible component models. Most of the code is written in Python classes so that slight variations can be developed with minimal effort, and so that multiple people can develop their own models without getting in the way of other people.

Backwards compatibility is important for people such as Robert who are using the simulations for their own research. The modular framework makes it possible for all of the models that Robert uses to continue to function while completely new master and component models are developed. When developing new models, consider whether you are better adding to an existing model, or if that model is already being used by others, whether adding a new model would be a better option. The existing users may for example like the simple functionality already offered by the model, and not want their simulations to change or break because you have a more complex or more realistic model to implement.

# Running the simulation using configuration files

## Configuration file format

### Master and sections

The configuration file defines which models are used in the simulation and sets configuration values for each model. The Master model is a special model that is responsible for hooking all of the other models together. As such, the ‘Master’ section name is fixed. All of the other sections may have whatever name you choose.

Comments are made with #. Any characters after the # on a line is ignored.

A section is specified by the line [section\_name] e.g. [Master], [Wind], above the values for that model. For example (a chopped-down file):

[Master]

model: master.txmultimastersimple.TxMultiMasterSimple

global: Global

iterations: 1000

algorithm: Algorithm

solar: Solar

dispatch\_order: solar

[Global]

timestep\_mins: 60

time\_period\_yrs: 10

carbon\_price: {2010: 20, 2020: 50, 2030: 75, 2040: 100, 2050:150}

[Algorithm]

model: algorithm.geneticalgorithm.Engine

base\_mute: 0.01

pop\_size: 50

[Solar]

model: generator.txmultivariablegenerator.TxMultiVariableGeneratorBase

capital\_cost: 1.0

data\_name: ts\_solar

Each section contains a set of lines of format parameter\_name : parameter\_value. Note that each section here, apart from the Global section, specifies a model. This model is the name of the Python class and the file it’s in, in its place in the directory structure. For example, the Algorithm model is found in algorithm/geneticalgorithm.py, class Engine.

For additional flexibility, the Master section does not directly expect particular section names for particular features. Instead, an indirect mapping is provided. Here, the Master defines ‘algorithm: Algorithm’, which means that when the master is looking for its algorithm, it looks in section ‘Algorithm’. If you wanted to swap algorithms, or the same algorithm but with different settings, you could define another section called ‘Algorithm\_2’, with whatever settings you wanted for that. Then, set the ‘algorithm’ parameter in the Master section to either ‘Algorithm’ or ‘Algorithm\_2’ as needed. The list of models that each master is looking for is discussed in the sections below about each master model.

Note that you can instantiate a model class in as many sections as you like. For example, the model class TxMultiVariableGeneratorBase is suitable for wind and solar generators. The same Python code is used but separate instantiations of the model are constructed.

### Single vs multi-period values

Some master models (of the TxMulti family) will make use of multiple configuration values per parameter. For each parameter, you can specify either a scalar value, as shown in the simple example above, or provide a set of values that map to different time periods. For example:

[Global]

carbon\_price: {2010:50, 2030:80, 2040:90}

When the configuration file is parsed, this is recorded as a Python dict. Then, the configuration for each model is replicated across all time periods of interest to that model to give a set of per-period configurations. The time periods of interest to the model are the union of the startup periods (any build time period listed in the startup data to the model), and the run periods (as specified in the master’s configuration parameter run\_periods). If no configuration is given for a particular period, the previous period’s value is used, or the first value in the list. For the example above, given startup periods of 1990 and 2000, and run periods of 2010, 2020, 2030, 2040 and 2050, the result in the configurations is as if it were specified:

carbon\_price : {1990: 50, 2000: 50, 2010: 50, 2020: 50, 2030: 80, 2040: 90, 2050: 90}

A software note – this config-expansion function is in tools/configurablebase.py in ConfigurableMultiBase.expand\_config. It currently requires at least one of the periods in the multi-period config to appear in the combination of the startup and run periods, but this could be (easily) fixed by writing more robust code.

### Parameter requirements and checking

Each model specifies the parameters that it requires, along with the type it will convert them to, and a default if sensible, in its get\_config\_spec() function, giving an object I’ll refer to as the ‘config\_spec’. The config\_spec functionality is discussed in more detail in the software structure section below.

You can find out what parameters the model expects by looking at the documentation (a Python docstring) to its get\_config\_spec() function. \*\*\*TODO\*\*\* write the docstrings for the master‼‼! \*\*\*\* Elly has written a script that collates all of these for all the models. Run, in the top-level of your checkout:

> python get\_config\_spec\_help.py

and you will see the Python docstring for each model that’s in the directory tree somewhere go flying past. The output is also written to the file get\_config\_spec\_help.txt for you to browse later.

The configuration system will check that all of the parameters listed, apart from those with defaults, are provided, and will raise an exception (halting the program) if not. It will log a warning if you provide a parameter that was not expected. This warning is useful to alert you to a probable typo, or that you are not using the model you expected. Some models, for example data/ncdata.py, will create their list of required parameters using some of the other parameter values you have specified. This will be detailed in the get\_config\_spec() help for that function.

### Global variables

All masters currently implemented support global variables. These are defined in the section pointed to by the ‘global’ parameter in the master, typically ‘Global’. These variables are available for any model that requires a variable of that name in its config\_spec.

The module tools/globalconfig.py calculates the following, which are widely used across the models:

* Converts **timestep\_mins** to **timestep\_hrs** and vice-versa. **timestep\_mins** takes priority if both specified. These parameters must have a scalar, not multi-period value.
* Calculates **carbon\_price\_m** from **carbon\_price** by multiplying by 1e-6, for the use of models that require the carbon price in $M/tonne instead of the specified $/tonne.
* Using the parameters **time\_period\_yrs**, **timestep\_hrs** and **data\_ts\_length** (where **data\_ts\_length** is set by the master once it has loaded the data timeseries), calculates the parameter **time\_scale\_up\_mult** that relates the time represented by the timeseries to the simulation time period.
* Calculates the parameter **variable\_cost\_mult** which is similar to **time\_scale\_up\_mult**, but could be (when implemented) configured with a discount rate parameter to account for the time value of money across the time period.

### Interaction between defaults, globals and configuration file(s)

The configuration of each model is built up in the following order:

* Default values, as specified in the model’s config\_spec.
* Global variables, taken from the global section for any variables specified in the model’s config\_spec.
* Values in that model’s section of the configuration files. There may be multiple configuration files specified, which are applied in order. The application of multiple files is discussed further in the section about command-line options below.

Note that when multiple time-period values are specified for a configuration value, if you overwrite the value using a later configuration file, this will overwrite the whole setting, not just the particular periods it specifies.

## Scripts to run simulations

### Command-line options

### Setting starting points

## Helper scripts for output processing

# Software structure

## Code documentation

### Python ‘help’ command

## Python packages

## Configuration system & Model framework

### Overall framework concepts

### config\_spec

#### Expanding config\_spec at runtime

## SimpleMureilMaster family

## TxMulti family

## Genetic Algorithm

## Data objects

### NetCDF reader

### Explicit data specification

## Generator objects

### Base classes

### Important considerations when implementing a generator

#### Handling of negative param values

#### Handling of ‘ts\_demand’ timeseries

### Variable generators

### Thermal generators

### Hydro generators

### Missed supply models

### Demand models

## Transmission models

## Master objects

### Output formats

## Multi-processing

## Unit and regression testing

### Unit tests

### Regression tests

### Running all tests

From the operating system prompt, run:

> python –m unittest discover –v

and expect it to take a few minutes. There will be some output as it runs. When finished, if all tests passed, it will report ‘OK’. If not, the reasons for the failures will be displayed. It’s good practice to run this before you check in code so that you know you have not broken some existing functionality. Note that there are a couple of tests that don’t pass for Roger for some reason – the test\_data/test\_ncdata.py and the test\_regression/rhuva\_test1. This is on the list of things to fix below.

# Web interface version (GE demo)

## Scripts to run GE demo

## Format of GE demo inputs and outputs

## Server implementation

# Further work

## Unfinished things

### Comments in geneticalgorithm.py on how it works

### Handling of negative params in gene

### Simple transmission model multi-period

Comment also that a simple-tx base class could be useful here.

### Using the get\_details function to identify demand and missed\_supply models in master

### Multi-period carbon handling in thermal models

### Globals for gas, coal prices, and handling in thermal models

### Complete the pumped hydro handling of multi-period, and for dam expansion

The comments in the code on what the units are of the parameters, in get\_config\_spec, also need clarification.

### Failing regression and unit tests

The test test\_data/test\_ncdata.py fails on Roger’s machine with the following message. It runs fine on Marcelle’s PC.

======================================================================

ERROR: test\_data.test\_ncdata (unittest.loader.ModuleImportFailure)

----------------------------------------------------------------------

ImportError: Failed to import test module: test\_data.test\_ncdata

Traceback (most recent call last):

File "/usr/local/python-2.7/lib/python2.7/unittest/loader.py", line 252, in \_find\_tests

module = self.\_get\_module\_from\_name(name)

File "/usr/local/python-2.7/lib/python2.7/unittest/loader.py", line 230, in \_get\_module\_from\_name

\_\_import\_\_(name)

File "/home/rogerd/MUREIL\_WC/test\_data/test\_ncdata.py", line 42, in <module>

import pupynere as nc

ImportError: No module named pupynere

The test test\_regression/rhuva\_test1 fails on Roger’s machine with the following message. It runs fine on Marcelle’s PC, and was set up from a simulation that Robert ran. The message isn’t very informative. It just says that the script single\_test.py that is in the test\_regression directory failed for some reason. A test\_out.pkl file wasn’t produced which suggests that it crashed somewhere. Further investigation is needed.

======================================================================

FAIL: test (test\_regression.rhuva\_test1.test.RegressionTest)

----------------------------------------------------------------------

Traceback (most recent call last):

File "/home/rogerd/MUREIL\_WC/test\_regression/rhuva\_test1/test.py", line 49, in test

test\_dir, config, pickle))

AssertionError: False is not true

### Cleanup of SVN branches

### Copyright messages

## Next steps

### Handling of discount rates

### Terminal values for models

### Calculation of O&M

### Capital cost models for multi-period

### Transmission model including flows

### Different dispatch order in different periods

### Multi-site thermal models

### Economic models

### Constraints on maximum total new build capacity

### Variable generators to use weather data instead of capacity factor data

## Ideas for performance improvement

### Genetic algorithm optimisations

#### Clone-test function performance

#### Combination with gradient-descent algorithm

#### Addition of a smaller-radius mutation

#### Definition of an ‘AlgorithmInterface’

### Orientation of timeseries data arrays

## Completion of formal testing

Some sections of the code are lightly tested, and/or not in the test set. Extra testing here would add confidence to the correctness of the models.

### timestep\_hrs

The models are written to accept the parameter ‘timestep\_hrs’ which specifies the timestep of the data timeseries. This is then used to calculate the MWh of electricity from a timeseries of MW, and the carbon emissions. Most of the use of the simulation to date has been with timestep\_hrs = 1.0. A specific review is needed of all models to check that timestep\_hrs is correctly applied, backed up with simple unit tests, and system tests where half-hourly and/or two-hourly data is used.

### Regression testing cleaning up and speeding up

The regression tests are currently a collection of whatever seemed to be an interesting test at the time, and together take a few minutes to run. Together they do cover a good proportion of the working functionality of the code. However, some of them take a long time to run. They could do with the number of iterations being reduced, with probably a minimal change to the effectiveness of the test. You can do this by editing the config file that’s in the test directory to change the iteration count, and then take the test\_out.pkl file and rename it to whatever the expected pickle file is. See the top of the test.py file for the name of the config and expected pickle files. Of course you can only do this with tests that already passed! The updated config and expected pickle files will be in SVN so make sure you commit them.

### Formal testing of the GE Demo results

# Python / System tips

## Performance Improvement

### Profiling

Profiling will help identify which parts of the program are taking the longest to run. Basic rule is - don't spend time optimising your code until you know what's taking all the time to run.

See: http://docs.python.org/2/library/profile.html#instant-user-s-manual

For example, run:

> python -m cProfile runmureil.py -f sample\_config.txt > sample\_config.prof

and browse sample\_config.prof to find where the time goes. The ‘cumtime’ column shows the total time spent inside this function and any functions it called. The ‘tottime’ column shows the time spent executing code actually in that function. Using sample\_config.txt as above you can see that 'tottime' for the calculate function is most of the run time of the sim. This is not surprising as this is the only calculate function that has a looped calculation in it - the others are all matrix maths which numpy does in a flash.

There are also ways to sort and search this information - see the help file for details.

### Numpy arrays vs Python lists

## SVN

SVN is the version control system on google code.

A list of useful commands here: http://www.thegeekstuff.com/2011/04/svn-command-examples/

The checkout instructions are on google code -> source -> checkout.

Most users will use 'add', 'commit', 'update', 'status' and 'diff'. It's good practice before doing a 'commit' to do 'status' and then do 'diff' on any files with an 'M' (for Modified) in front of them, to be sure you know what you've changed.

If you do an 'update' and it says that the merge failed, the file will be in conflict. SVN tries to combine changes that someone else has checked in with changes that you may have made locally. If you edit different parts of the same file this is likely to work. If you have edited the same parts of the file, then it will report a conflict.

See here for how to resolve it:

http://www.websanova.com/tutorials/svn/svn-conflicts

Don't whatever you do choose the (mc) mine-conflict option if 'update' offers you that. What that will do is ignore whatever you just updated and just use your new version - so you may be throwing away someone else's edits. This is often hard to find out and makes people very cross! (p) postpone is the best option.

## Finding Python help

Google is your best friend here. If you start a question with ‘numpy’ or ‘python’ you’ll get a good response. The site stackoverflow.com will often produce very useful suggestions and code snippets, with commentary.

The Python tutorial at <http://docs.python.org/2/tutorial/> is generally helpful and a good introduction to lists in particular.

The Numpy tutorial at <http://www.scipy.org/Tentative_NumPy_Tutorial> is essential reading to understand and use Numpy.

Also have a look at Scipy if you want to find library functions for a wide range of applications, in particular see scipy.optimise <http://docs.scipy.org/doc/scipy/reference/optimize.html>.

## Random Python tips

### Splitting lines

# Old stuff

This is a good place to copy stuff from the document (e.g. from the Further Work section) that’s not current, but might still be interesting to someone.