**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. It is assumed that the reader has basic familiarity with Python, though the configuration files can be used to set up and run simulations without any knowledge of Python.

**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 (i.e. classes defined in Python) 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 wind gas

[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. For parameters where the value is a list of values, such as for the ‘dispatch order’ in the Master section above, enter them with spaces between the values. For parameters where the value is a multidimensional array, such as for model startup values, use the Python array syntax, e.g. startup\_data\_string: [[55, 2000, 2010], [44, 1990, 2030]].

Note that each section here, apart from the Global section, specifies a model (the Global by default is the tools.globalconfig.GlobalBase 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() method, 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() method. 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. This file is an excellent reference for the functionality and configuration of all models in the system.

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 model.

### 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 model tools.globalconfig.GlobalBase 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.
* Finally any command-line options are applied.

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

### runmureil.py

The basic script to run simulations is **runmureil.py**. A basic simulation needs only a configuration file to be provided:

> python runmureil.py -f asst5\_config.txt

will give output:

CRITICAL : Run started at Fri Mar 08 11:39:47 2013

INFO : Interim results at iteration 0

INFO : best gene was: [964, 4993, 588, 2200, 2221, 7452]

INFO : on loop 0, with score -330375.190080

INFO : wind ($M 249220.00) : Wind with capacities (MW): 5880.00 22000.00 22210.00 74520.00

INFO : solar ($M 59570.00) : Solar\_Thermal with capacities (MW): 9640.00 49930.00

INFO : fossil ($M 21585.19) : Instant Fossil Thermal, max capacity (MW) 6019.80

INFO : Total cost ($M): 330375.19

INFO : Interim results at iteration 500

INFO : best gene was: [863, 345, 220, 832, 1742, 2257]

INFO : on loop 465, with score -190220.501680

INFO : wind ($M 101020.00) : Wind with capacities (MW): 2200.00 8320.00 17420.00 22570.00

INFO : solar ($M 12080.00) : Solar\_Thermal with capacities (MW): 8630.00 3450.00

INFO : fossil ($M 77120.50) : Instant Fossil Thermal, max capacity (MW) 16421.30

INFO : Total cost ($M): 190220.50

CRITICAL : Run time: 8.38 seconds

INFO : best gene was: [854, 388, 83, 850, 1764, 2163]

INFO : on loop 976, with score -189399.751200

INFO : wind ($M 97200.00) : Wind with capacities (MW): 830.00 8500.00 17640.00 21630.00

INFO : solar ($M 12420.00) : Solar\_Thermal with capacities (MW): 8540.00 3880.00

INFO : fossil ($M 79779.75) : Instant Fossil Thermal, max capacity (MW) 16723.80

INFO : Total cost ($M): 189399.75

### Command-line options

The command line options are processed (and documented) in the function tools/mureilbuilder.py:read\_flags. The docstring for this function explains how to add new options. Each option, apart from the logging options, overrides corresponding values set in the configuration files.

The options to **runmureil.py** are:

**-f configuration\_filename**: You can have as many of these as you like. They will accumulate the configs with later files taking precedence. See 'batch files' below for an example of this.

**--iterations count**: Set the number of iterations to do – e.g.--iterations 2000. Overrides ‘iterations’ in the Master section.

**--seed seed**: Set the random seed – e.g. --seed 12345. Overrides ‘seed’ in the algorithm section.

**--pop\_size size**: The size of the gene population – e.g. –pop\_size 100. Overrides ‘pop\_size’ in the algorithm section.

**--processes count**: How many processes to spawn in multiprocessing. Set to 0 to do no multiprocessing. Overrides ‘processes’ in the algorithm section.

**--output\_file filename**: The name of the output file. Overrides ‘output\_file’ in the Master section.

**--do\_plots {False,True}**: Set to either False or True. If True, print plots at the end of run. Overrides ‘do\_plots’ in the Master section.

**--run\_periods period\_list:** Overrides ‘run\_periods’ in the Master section. Surround period\_list with double-quotes – e.g. --run\_periods “2010 2020”

### Logging

Command-line options are provided for configuring the logging. These are handled in do\_logger\_setup in tools/mureilbuilder.py. A single logfile is maintained across the simulation. If no logfile is specified, output will be to the console as stderr.

**-l filename:** Filename for a log file. If not set, output will be to the screen (via stderr).

**-d debuglevel:** One of DEBUG, INFO, WARNING, ERROR, CRITICAL. INFO is recommended and is the default.

**--logmodulenames:** if present, log the name of the module that the log message is sourced from. This makes the logfile large and hard to read so is off by default.

To make a logger output from your model, add the following lines to your Python file, above other code:

import logging

logger = logging.getLogger(\_\_name\_\_)

and then you can call the standard Python logger module functions, for example:

logger.critical('Run started at %s', time.ctime())

### Setting starting points from previous results

To initialise the whole population to identical gene values, read from a file, use runmureil\_gene.py. Run as follows:

> python runmureil\_gene.py test.pkl

where test.pkl has been saved from a previous run (or you have created it), so it contains a dict with 'best\_gene' or ‘best\_params’ as a member.

### Batch scripts – runmulti.py

See **runmulti.py** for an example of a batch processing script that creates a bunch of extra config files for a series of values of a parameter, runs a simulation with these files in addition to the base config file, then collects the results. This makes use of the existing **runmureil.py**.

Note that you can update almost any of the configuration script values in this way (see exception for Master below) - including 'model'. The technical operation of this stacking of configuration files is that the Python dict built for that configuration file section has ‘update’ called on it with the new settings. This will override any that were already there, and add any new ones.

The master’s names for the configuration file sections cannot be changed in subsequent files as the section names are used to accumulate the values across files.

## Helper scripts for output processing

The following scripts are in the mureil-ga directory. Run at command prompt:

> python plotpickle.py pickle\_filename

to draw plots of cumulative and separate power timeseries. (Note this has not been updated to work with multi-period simulations).

> python printpickle.py pickle\_filename > output\_filename

to dump the pickle file to a text file.

# Software structure

## A note about Python packages

Python packages are used in this simulation to help organise the code and make it easy to refer to code from other subdirectories. Every subdirectory is a ‘package’. You can look up in the Python documentation what packages are all about, but really what you need to know is:

* Use ‘.’ to identify subdirectories, not ‘/’
* When referring to a module, always refer to it relative to the base directory of the checkout, e.g. to import the txmultigeneratorbase module into any file, anywhere in the hierarchy, type this in your code:

from generator import txmultigeneratorbase

* If you want to load just the module you’re interested in within the Python interpreter, start the interpreter from the base directory of your checkout and then load the module you want. If you try to load the module just on its own then it won’t load other modules it depends on correctly. For example:

import data.ncdata

* Always put an empty file called \_\_init\_\_.py in each subdirectory. This is a requirement of Python – something to do with identifying that directory as one that contains package modules. You’ll see the \_\_init\_\_.py files scattered all through the code.

## Code documentation

Most of the code is documented in-file using Python docstrings. The script get\_config\_spec\_help.py will crawl the directories and collect docstring information (class docstring and the get\_config\_spec method docstring) from all classes subclassed from ConfigurableBase. For users of these models, the docstring on the get\_config\_spec method is important as it describes all the parameters the model expects, so be sure to provide useful information here.

Python has a nifty ‘help’ function which neatly displays all the docstrings for a module, and the docstrings for classes further up the hierarchy. This means that when you run ‘help’ on your model module of interest you should see the documentation on all the functions, not just those specifically defined in that file. To use help from within Python:

>>> import data.ncdata

>>> help(data.ncdata)

Or, from outside Python, use a little Python script **showhelp.py**:

> python showhelp.py data.ncdata

or shell script:

> ./showhelp.sh data.ncdata

Note that I have used the package path not the file path here to identify the module. It would be more convenient if the script would take a file path and then make the package path from it, as then the tab-completion on the filename would work.

## Configuration system & Model framework

### Starting the simulation

The script **runmureil.py** begins by building a master instance using build\_master in tools/mureilbuilder.py. This calls a series of functions in mureilbuilder.py (which are well documented in-file) that parse the configuration file(s) and command-line options, then create the master model, then call the master’s set\_config method with a Python dict that includes all of the sections in the configuration. The set\_config method then creates instances of all of the other models in the configuration and sets them up together.

Once the master is created, runmureil.py calls ‘run’ on the master. Finally (whether or not an exception occurs), the master.finalise method is called. This finalise method is important when multiprocessing is enabled as it allows the multiprocessing to be gracefully exit at the end of processing.

### Configuration Base classes

The framework relies on all parts of the model being configurable with the configuration file. A generic interface class ConfigurableInterface is defined in tools/mureilbase.py. An implementation of this (used by all models) is ConfigurableBase, implemented in tools/configurablebase.py. For multi-period simulations a subclass ConfigurableMultiBase adds functionality to define and hold different configurations for different periods. All models are required to subclass from ConfigurableBase or ConfigurableMultiBase. These operation of these classes are best understood by reading the code (with comments) in the file.

ConfigurableBase implements the method ‘set\_config’ as a series of calls to other methods. Any of these methods may be overridden by the model. For example, the method ‘complete\_configuration’ which is the last in the chain, and empty in this base class, may be overridden by a model class that pre-calculates some of its working values from the configuration values. For example, the Data class in data/ncdata.py uses the complete\_configuration method to load in all the data from the NetCDF files based on the configuration.

### config\_spec

The ConfigurableBase and ConfigurableMultiBase classes make use of many helper functions defined in tools/mureilbuilder.py. Many of these revolve around the ‘config\_spec’ object. The config\_spec is a list of tuples that specify what the parameters are for each model. Models return a config\_spec object from their get\_config\_spec method, and then assign it to self.config\_spec.

The format is (parameter\_name, conversion\_function, default\_value).

**parameter\_name** is a string.

**conversion\_function** is any unary function (a function that takes one argument). Functions specified include ‘int’ and ‘float’, and a number of other functions found in mureilbuilder.py that handle lists of strings, lists of integers, and boolean values. The functions should be written such that they will convert a string value (as read from the configuration file) to the desired type, but they will also let the value pass through unaltered if it is already of the desired type. See the make\_int\_list function in mureilbuilder.py for an example. If the desired type is string, set the conversion function to None.

**default\_value** is either a string or a value in the desired type. The conversion function will be applied to the default value. If default\_value is None, no default is assumed.

mureilbuilder.py provides functions to check if all of the parameters in the config\_spec are present in the configuration, and if there are any extras. These functions are called from the ConfigurableBase and ConfigurableMultiBase set\_config methods.

#### Expanding config\_spec at runtime

In some models it is useful to determine what the full parameter list is from other parameters. For example, the Data model in data/ncdata.py wants to know the list of the data variables it is to read in, and then for each of these it requires the further information of the filename, and optionally the variable name within the file. The ConfigurableBase method process\_initial\_config is designed to be overridden by models that build the config\_spec at runtime. Refer to ncdata.py for an example.

## 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

The gene params are typically interpreted as the amount of capacity to install at each site. This is fine if you want to be strongly biased towards building at all sites – however if you don’t, you want the algorithm to choose between building and not building with some reasonable probability. One way to do this is to encode negative gene values to mean ‘don’t build’. The min\_param\_val could be set to –max\_param\_val, if desired, or even smaller. What’s important is that the models that interpret these gene values are aware of the meaning of negatives. The update\_state\_new\_period\_params method in TxMultiGeneratorMultiSite, used by most of the TxMulti family of generators, ensures the capacity is set to zero for any negative gene param values. This may not be the case for the SinglePassGenerator models and should be checked in these.

#### 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

The GE demo (the back-end of the GE-sponsored energy-system-building interactive website) is run from **model\_web.py** if you are a webserver, and **model\_file.py** if you want to test it locally.

**model\_file.py** and **model\_web.py** set up a couple of parameters and then call **rungedemo.py**. rungedemo.py is very similar to runmureil.py discussed previously.

The first cut of the GE demo included some major hacks, mostly to do with trying to put in multi-period functionality where it didn’t exist. The latest version uses the TxMulti family of master and generator models and is much neater.

The file GEconfig.txt is an important part of the system. It sets the list of generator types in the dispatch order to that expected by the web interface. Note that the slow-response thermal model is not currently enabled – but it does now exist and should be used.

A sample data file, for one site only and only two months, is used by GEconfig.txt. Data for multiple sites could be added once the method find\_site\_index in GeTxMultiMaster is filled in.

## Format of GE demo inputs and outputs

The format here is as interpreted from the sample files provided by the web interface designers. It’s not difficult to change the Python code if requirements change.

The JSON format is used for both inputs and outputs. Example inputs are:

{

"selections": {

"generators": [

{

"id": "coal\_1",

"type": "coal",

"lat": -38.42,

"long": 144.17,

"decade": 2000,

"decomission": 2050,

"capacity": 150

},

......

such that a call to json.loads(json\_data)['selections']['generators'] yields a list of generators, as shown above, and later in the input stream:

"demand": {

"2010": {

"residential\_efficiency": 50,

"residential\_intelligence\_transmission": 50,

"residential\_micro\_grids": 50,

"residential\_medium\_scale\_distributed": 50,

......

"grid\_small\_scale\_solar\_pv": 50,

"grid\_demand\_management": 50,

"grid\_storage": 50,

"grid\_smart\_meters": 50

},

such that a call to json.loads(json\_data)['selections']['demand'] yields a dict of periods, where each value contains the set of configuration settings for the demand model.

Outputs are a JSON stream, from a call to json.dumps on a results dict (somewhat reorganised from the simulation results):

{ '2010': { 'co2\_tonnes': 334745997.5300526,

'cost': { 'coal': 11716.109913551843,

'demand': 0,

'gas': 0.0,

'missed\_supply': 0.0,

'solar': 0.0,

'wind': 503.75999999999993},

'demand': '35109501.67',

'discounted\_cumulative\_cost': 12219.869913551844,

'output': { 'coal': '33474599.75',

'demand': '-35109501.67',

'gas': '0.00',

'missed\_supply': '0.00',

'solar': '0.00',

'wind': '1634901.92'},

'period\_cost': 12219.869913551844,

'reliability': 100.0},

'2020': { 'co2\_tonnes': 431094804.37514085,

......

where (as also documented in model\_file.py and model\_web.py):

* **cost:** summed per generation type, total for that decade, in $M
* **output:** summed per generation type, MWh per year
* **reliability:** the percentage of hours where demand was not met
* **co2\_tonnes**: in tonnes, total for that decade
* **period\_cost**: total of all 'cost' values, for that decade, in $M, without discounting.
* **discounted\_cumulative\_cost:** total of all 'period\_cost' values including the current decade, with discounting at the configured discount rate assuming all costs incurred at the start of the decade.

## Server implementation

The initial implementation initiated a new Python session for each update to the interface. This is likely to be very slow when a real-size dataset is used. A more efficient approach would be to load up the data and configure the simulation once only. There are comments in rungedemo.py that indicate how this split would be done. The GeTxMultiMaster is structured so that this is straightforward.

It is important to note that while the setup and run may split neatly into parts, the run must still be called in a sequential fashion. One request must be finished before the next request is commenced, as the code (calc\_list\_cost in GeTxMultiMaster) that applies the demand settings modifies the self.period\_configs variable and so is not thread-safe.

# Further work

## Unfinished things

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

geneticalgorithm.py is lacking commentary on how the algorithm actually operates. I’ve taken a guess at what the configuration parameters mean but I’m not really sure.

### Handling of negative params in gene

As discussed in the section on negative gene params above, the SinglePassGenerator models need to be checked to make sure they sensibly handle negative gene param values.

### Simple transmission model multi-period

Roger has implemented a very simple transmission model (transmission.distancetxmodel.DistanceTxModel) that charges to build the transmission line to the nearest trunk node. This model works in a single period, but it does not identify whether the line was already built in an earlier period. Some memory of the previous period’s state needs to be added to this model.

For best software practice, a base class for simple transmission models of this type could be implemented, and then checked for by the master (using the check\_subclass function in mureilbuilder.py), so that different transmission models could be specified interchangeably in the configuration file.

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

The TxMultiGeneratorBase class defines a get\_details method that returns among other things what type of model it is, specifically identifying demand-source, demand-management and missed-supply models. These flags could be used to sensibly plot the results and identify data of interest for output. Currently the master models identify demand-type ‘generators’ by their name of ‘demand’ to the master. Using the get\_details method would be more robust.

### Convert slowresponsethermal\_beta.py to TxMulti format

Elly has written the slow response thermal model using Michael’s description of the beta thermal model. This was written based on the SinglePassGenerator class. To be used with the TxMulti masters it needs to be converted. An example of such a conversion is the txmultislowthermal.py module. Note that this is not a large task if you start with txmultislowthermal as a template.

### Multi-period carbon handling in thermal models

The thermal models in the TxMulti family, in thermal/txmultislowresponsethermal.py and txmultiinstantthermal.py handle a single site only, but they can model generation capacity being added in different periods. It is conceivable that such capacity would vary in its carbon intensity, or fuel use or whatever else. The model has the configuration for each installation period available to it for use in calculation, but currently just uses the values for the current period. Code to divide up the total output between the different periods is required so that the corresponding carbon and other configuration can be used.

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

It would be nice to be able to provide a global ‘gas\_price’ and ‘brown\_coal\_price’ etc. The thermal models are currently generic, so could not expect a parameter for anything other than ‘fuel\_price’. The config\_spec could be updated at runtime (see the section above on updating the config\_spec at runtime), based on the name of the fuel. For example, add a config\_spec parameter ‘fuel\_type’, e.g. ‘gas’ and then use this to construct a config\_spec entry requiring ‘gas\_price’. The global parameter could then be extracted. There is a complication here in that the ConfigurableBase code does not re-apply the global values when it reprocesses the config\_spec in update\_from\_config\_spec as doing so would overwrite values from the configuration file for existing parameters. A smarter update from the global parameters in this method would need to only set values for parameters that were not yet in self.config.

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

The pumped hydro models don’t completely handle multi-period as the dam starting level is reset at the start of each period. The comments in the code on what the units are of the parameters, in get\_config\_spec, also need clarification.

Expansion of the dam could also be modelled in the pumped hydro model. This could be done by defining additional optimisation params that represent the building of dam capacity, and some overriding of methods of TxMultiGeneratorMultiSite to interpret the extra params as dam capacity rather than electrical capacity.

### 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

The ‘roger’ and ‘gedemo’ branches are out of date and should be deleted.

### Copyright messages

The copyright messages say 2012. They could be updated to 2013 using Roger’s add\_copyright.pro script.

## Next steps

### Handling of discount rates

Discount rates are important in a multi-period simulation. This is currently handled in a bit of a hacky way for the GE demo. It could be incorporated into the TxMulti system in several places:

* The **variable\_cost\_mult** parameter. Say that you have 2 years of data, but want to extrapolate to 10 years. The calculation, assuming no discounting, is already done – but you should incorporate a discount factor in here as well. This would need to go into the global value calculation in tools.globalconfig.GlobalBase.
* In TxMultiMasterSimple calc\_cost method. The line cost += period\_cost should be modified to account for discounting.
* In TxMultiGeneratorMultiSite calculate\_time\_period\_simple method. The decommissioning cost, considered incurred at the end of the period, as the capacity is used throughout the period, should be discounted for a period length before adding to the total cost.

### Terminal values for models

The multi-period cost calculation currently bills you for building the capacity, but only gives you output for the periods modelled in the simulation. This means there is quite a disincentive to build capacity in or near the final periods, and so non-intuitive optimisation results may be seen.

There are hooks in the code for a get\_terminal\_value function that calculates the terminal (or residual) value at the end of the specified period. This code is just waiting to be filled in. If discounting is not used, a very simple approach would be to divide the capital cost of the installation over the number of time periods of its life and count how many periods remain. This applies the principle that the value of the installation is the NPV of all future revenues. This is more complex to calculate when discounting is involved.

### Calculation of O&M

O & M costs (operating & maintenance) are not modelled yet, but could be added by adding configuration parameters to models to describe how the O & M is incurred, and adding the calculation of them to the model’s calculate\_outputs\_and\_costs method.

### Capital cost models for multi-period

Robert implemented a number of complex models of how capital cost related to number of wind turbines installed in his single-period models, assuming there was no existing installation. For a multi-period system there is the possibility of existing and/or recently decommissioned capacity to consider. The TxMultiGeneratorMultiSite class provides a simple model, linear in capacity, that charges an extra installation cost if that’s the first time the site is used. The method calculate\_capital\_cost\_site is available for overriding, and has access to details of current and past capacity, so costs of any desired complexity can be implemented, and probably needs to be, to make a proper estimate of the costs.

### Transmission model including flows

The TxMultiGeneratorBase class defines a method calculate\_time\_period\_full that exposes most of the internal calculations of the generator. A transmission model that calculated power flows would need to control the dispatch, and do the summing of power, costs and carbon that the calculate\_time\_period\_simple method does within the generator model. This is quite different to what the ‘simple’ masters currently do.

I have implemented a rough idea for an iterative power-flow model by providing a ‘max\_supply’ argument to the generator, so the transmission model can implement curtailing, and a recalculate\_time\_period\_full method to recompute the variable costs and the output when the supply request and curtailing changes. A ‘price’ argument is also provided (rather optimistically!). See the docstring for recalculate\_time\_period\_full to see how to correctly call the methods in turn.

### Different dispatch order in different periods

This is straightforward. Currently the TxMultiMasterSimple model uses the dispatch\_order value to determine which generator models it needs to instantiate. It could just as easily compile the list of required generator models from a multi-period setting, and select the dispatch\_order from self.period\_configs instead of just using the one order from self.config. The line in the configuration file would look something like this:

dispatch\_order: {2010: solar gas coal, 2020: solar wave gas}

### Multi-site thermal models

The thermal models in the TxMulti family restrict themselves to a single site so that there is no question of how to dispatch the capacity to meet the supply request. Multiple sites can currently be modelled by listing them in the master’s dispatch order and configuring them individually. The thermal models could be extended to handle multiple sites once a method for dispatching between the sites was determined.

### Economic models

An economic model of the profitability of each site could be constructed using the calculate\_time\_period\_full method of TxMultiGeneratorBase. This exposes the per-site information on output, capital costs and variable costs. The master (or transmission model, if it was doing the sums) could direct the relevant per-site information to an economic model.

### Constraints on maximum total new build capacity

There is a concern that the optimiser may come up with solutions that require a very uneven distribution of manpower and resources across the time periods. One simple way to model this would be to have the period\_cost value (the total expense for the period), as currently calculated in TxMultiMasterSimple, to be transformed by a function that increased costs more than linearly as the total period cost became unreasonable.

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

The variable generators currently expect the timeseries data to be provided as capacity factor. Robert calculates the capacity factor data from the meteorological data and power curves of typical equipment. A more configurable arrangement would be to have the meterological data loaded from files and the variable generator configuration to include some specification of a conversion function. This would have the disadvantage of needing to do all these calculations when the data was first loaded, a significant issue for large data sets.

## Ideas for performance improvement

### Genetic algorithm optimisations

#### Clone-test method performance

Profiling of Robert’s simulations that had large gene populations showed that the clone\_test method in geneticalgorithm.py was consuming a substantial proportion of the run time. Effort into optimising the performance of this function, particularly in reducing how the runtime increases as the population size increases (the big-O of the algorithm), will be well worthwhile.

#### Addition of a smaller-radius mutation

The values in the genetic algorithm change in their combinations but do not change value when breeding occurs, and mutation (controlled by the base\_mute parameter) picks a new value from the full range available. This means that while the algorithm is great at trying lots of different regions of the search space, it’s slow to find a more optimal solution that is very near in value to the current best gene, as a better value has to be lucky enough to come up.

I have implemented a smaller-radius mutation to assist the algorithm in performing a more gradient-descent like function. This is documented in the get\_config\_spec in geneticalgorithm.py as ‘local\_mute’. I don’t know what the ideal settings would be. I ran some rudimentary tests on a simple configuration with a TxMultiInstantOptimisableThermal, where there’s a fairly continuous and smooth gradient, and found that the optimisation found a smaller cost in about half the time when the local\_mute parameter was used. Without the local\_mute, the optimisation got stuck with the same best result for numerous iterations.

An aside – I added this parameter, but made the code backwards-compatible in Pop.mutate() by checking that the local\_mute parameter was non-zero before calling random.random. This means that the order of the random values was not disturbed, so existing tests would give the same results as before to the same random seed.

#### Definition of an ‘AlgorithmInterface’

Similar to what is done with the DataSinglePassInterface (defined in tools/mureilbase.py, and checked for by the master), an AlgorithmInterface could be defined to allow for safely interchanging algorithms.

### Orientation of timeseries data arrays

This is a bit of a confusing one, as IDL and Python define multi-dimensional arrays differently. For a two-dimensional array, in Python the second index refers to the fastest-changing memory address. In IDL it’s the other way around. So I’m not actually sure what the NetCDF reader does when it reads in the timeseries data arrays. Python’s numpy array is friendly in allowing an array to be viewed either way (you can call ‘transpose’ on an array without changing any data in memory), but I’m not sure what this means for the speed of processing. My particular concern is that we do lots of dot-product calculations per-site with the variable generator data, but the data comes from the NetCDF as per-timestamp. I imagine that in memory we have all the data for each timestamp grouped together, so to do the dot product the numpy calculations have to jump more than one memory location each time, which may or may not be a performance issue. I haven’t tested this with a big dataset. Backwards compatibility needs to be considered if you choose to flip the orientation of the array as required in the NetCDF.

## 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

The GE demo output results have not been comprehensively verified. The input file used in the regression test (in test\_regression/gedemo1) is not very interesting. A more detailed test, with more variety in the input values, and with hand (or spreadsheet) computed expected results, will give more confidence.

# 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

Both numpy arrays and Python lists are used throughout the code. Numpy arrays are specifically written to crunch large amounts of data quickly – for example sum and dot-product operations on large arrays are enormously faster than a similar calculation for a Python list. Python lists are designed to be fast for adding and removing elements from. Numpy arrays are particularly poor at adding and removing elements as all the memory is reallocated and the data copied each time.

## 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

You can split a line in Python if the expression being split is within brackets. For example:

x = numpy.array(

[1, 2, 3])

This also works if you are within square or curly brackets (i.e. already within a list or dict). If you want to split a line but aren’t within brackets, you can add a set of extra round brackets around any expression.

# 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.