Mureil Python Software

1 Introduction

1.1 Document Summary

Author: Marcelle Gannon – mgannon@student.unimelb.edu.au, 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,Michael Brear, Elly Hutton, Steven Thomas, Peter Rayner.

1.2 Outline of modelling approach and history

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

2 Running the simulation using configuration files

2.1 Simulation structure outline

The simulation is implemented by hooking up and configuring models which are Python classes. A 'Master' model is chosen by the configuration file, and instantiated first when the simulation starts. It then instantiates and configures the other models. The other models fall into 5 categories which are explicitly handled by the master.

Two types of master model have been implemented. Both types expect the other models to perform calculations on the full data timeseries at once. The framework as implemented does not cater explicitly for interaction between generator models at each timepoint, as might be required to do spot market pricing.

The first version was the 'SimpleMureilMaster', which was suited to single-period simulations. Generators of type SinglePassGenerator are compatible with SimpleMureilMaster.

The TxMultiMasterSimple model is suited to multi-period simulations. It also adds a system for identifying multiple sites explicitly, so a transmission model can work with it. The name 'TxMulti' is short for 'Transmission-Multi-Period'. Generators of type TxMultiGeneratorBase are compatible with TxMultiMasterSimple. It is suggested that models that fit this TxMulti framework are used for future model and configuration development. When required, a single-period simulation can be run by selecting only a single period to run.

2.1.1 Global

These are models to collect and do calculations on global variables, which are then available to other models. Only one (GlobalBase) is currently implemented.

2.1.2 Algorithm

These are classes defining the algorithm that searches for the lowest-cost system by optimising a vector of values (hereafter often referred to as the 'params', as distinct from 'configuration parameters'). The master will supply the algorithm model with a cost function which the algorithm calls. Only one algorithm (geneticalgorithm.py:Engine) is currently implemented.

2.1.3 Generator

These are any model that represents a generator that contributes to the supply (or consumption) of power, and calculates a cost for this. This includes demand and missed-supply models. The master typically holds a collection of these. Generators may represent multiple sites with the same configuration but different weather data and installed capacity.

2.1.4 Data

Data models provide all the vector and matrix input data to the simulation. These may be hard-coded (for example data/mg_sample_data.py), or fully configurable (data/ncdata.py). A data model is based on the DataSinglePassBase class. The data model will return a vector or matrix of data when requested by name. Both of the current master models implemented handle only one data model, but conceivably a combination of models could be used.

Each model in the simulation is able to request (through its get_data_types method) the data series it requires, and the master will deliver it via the model's set_data method. The documentation for each model will indicate what the names of the required data series are and/or how the names of the data series are chosen. For example, the TxMultiVariableGeneratorBase model, used for weather-dependent generators, takes a configuration parameter 'data_name' that specifies the name of the required data series. When writing the configuration file, the user needs to ensure that the data series name provided to 'data_name' is what the data model provides. The ncdata model is described in more detail further on.

2.1.5 Main functional loops - Sequential dispatch model and multi-period operation

Both the TxMulti and SimpleMureil master models implement a sequential dispatch model. This takes the configuration parameter 'dispatch_order' and builds its generator collection from this. To calculate the cost and output of the system, a demand timeseries is started with, and each generator in turn is presented with the remaining demand as a supply request, for the full timeseries at once. If the name 'demand' appears in the dispatch_order, the initial demand is set to zero and the demand model is assumed to provide some demand (as negative supply). Otherwise the demand is loaded from the ts_demand series in the data model.

For multi-period operation with the TxMultiMasterSimple, the outer loop is on the period. For each period in turn, the capacity of each of the generators is updated, demand dispatched using the sequential dispatch model, and the transmission model asked for a cost. Costs are then aggregated across periods. Section 3.5.1 discusses how the generator model state is maintained across periods.

See section 5.2.6 for details on how this dispatch order could vary between periods, and section 5.1.4 for how the demand timeseries could be done more neatly in the TxMultiMasterSimple than by looking for 'demand' in dispatch_order.

2.1.6 Optimisable param mapping

The optimisation algorithm works on a list of param values. These are built up from the requirements of each of the generators, and for the TxMulti master the set is repeated for each time period to build up a total list to be optimised. Typically these params represent new capacity to install at a site in that time period, but they could be used for any purpose.

2.1.7 Site indices and transmission model

All TxMulti generator-type models maintain a list of site indices, and return a list of sites with active capacity to the TxMulti master. The TxMultiMasterSimple requires one or zero transmission models. It provides a list of sites with active capacity to the transmission model and the model calculates a cost from this. More complex transmission models could be fitted in – see the later section 5.2.5 about this.

Generator models that handle only one site will typically take a parameter called 'site_index' that defines the site they represent. Models that handle more than one site (for example TxMultiVariableGeneratorBase) need to have the list of sites configured. A configuration parameter params_to_site_data_name specifies the data series (see section 2.1.4 above) that maps the params taken from the optimiser for this generator to the site index they represent. This is simply a vector of integers. The configuration parameter params_to_site_data_string allows for the list to instead be specified directly in the configuration file.

The variable generators also take a mapping between the site index and the column of timeseries data to apply for that site. A one-to-one mapping with the params_to_site_data setting described in the paragraph above is the default, but it can also be configured. The configuration parameter data_map_name identifies the data series

holding the mapping. This data series is a 2 x n mapping of [[site1, data1], [site2, data2] …], where the data numbers are the columns the weather data matrix.

See the get_config_spec help (refer to section 2.2.3) for how to look up the help for each generator. This help will indicate which settings are required for the param-site and site-data mappings.

2.1.8 Startup states

The TxMulti type models support startup states. The startup state represents capacity installed for that generator in periods before the run periods. In models based on TxMultiGeneratorMultiSite, the startup state can be provided by a data series, a configuration parameter, or both. The startup state is essentially a list of (site, capacity, build-date, decommission-date). The documentation for TxMultiGeneratorMultiSite explains the format of these.

Within a multi-period configuration setting, values for periods referred to in the startup state list can be provided and will be used by the generator models if required for the calculations. For example, the carbon intensity of an old plant may be much higher than the new installed capacity.

Startup capacity may be installed at sites that are not available for new capacity. This is done by simply not listing the site index in the params_to_site_data settings.

2.2 Configuration file format

2.2.1 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.

2.2.2 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.

2.2.3 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 the model does and 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.

2.2.4 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.

2.2.5 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.

2.3 Scripts to run simulations

2.3.1 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
```

2.3.2 Multi-period simulation

A simple multi-period example simulation is described in asst5_config_multi.txt. The same runmureil.py script as before is used, as all the information is encoded in the configuration file:

```
> python runmureil.py -f asst5_config_multi.txt
```

This will give you lots of interesting output! The file asst5_config_multi.txt also shows how to set up the param-site and site-data mappings.

2.3.3 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"

2.3.4 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())
```

2.3.5 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.

2.3.6 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.

2.4 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.

3 Software details

3.1 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.

3.2 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.

3.3 Configuration system & Model framework

3.3.1 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.

3.3.2 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 class. The functions that set_config in ConfigurableBase calls are, in turn:

- **load_initial_config:** loads defaults, then applies config and global, checks all required present
- **process_initial_config:** empty here - may determine extra parameters required and update the config_spec
- **update_config_from_spec:** process an updated config_spec
- **check_config:** check all parameters for existence, and that there are no extras

- **complete_configuration:** empty here, to override in classes that do further processing

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.

The ConfigurableMultiBase class calls the following functions instead of complete_configuration. This allows for further processing to happen either before or after the configuration is expanded into the set of configurations for each period.

- **complete_configuration_pre_expand:** empty here, to override in classes that do further processing
- **expand_config:** fill out the period_configs dict
- **complete_configuration_post_expand:** empty here, to override in classes that do further processing

To handle a multi-period configuration value before expansion into multi-periods, the code needs to determine if it is a single value or a multi-period value. A multi-period value will be stored as a Python dict, and any processing needs to be done on each value. See thermal/txmultiinstantthermal.py:TxMultiInstantOptimisableThermal for an example.

3.3.3 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.

3.3.3.1 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.

3.4 Data models

The use of the Data models is described in section 2.1.4. Note that the timeseries data are expected to have the timeseries as the first index and site as the second (in Python array-indexing language), not the other way round. See

data/mg_sample_data.py for an example. This reflects the layout of the data in the source. Section 5.3.2 discusses whether this is the best orientation for fast running.

3.4.1 NetCDF reader

The data model data.ncdata.Data provides a generic NetCDF reader. The get_config_spec docstring explains the configuration parameters. In summary:

- User configures a list of desired data series names, in the categories of 'ts_float_list', 'ts_int_list', 'other_float_list' and 'other_int_list'.
- For each data series, provide a NetCDF filename and variable name for each. By default the variable name matches the data series name.
- All 'float' arrays are converted to numpy.float64 type, and the 'int' arrays to numpy.int32 type.
- The data series in the ts_float_list are checked for NaNs (note that an int32 cannot be NaN), and any timepoints found with an NaN are dropped out across all of the timeseries data sets.
- A check is made that all timeseries are the same length, and this length is made available in the get_ts_length method, for the master to request.

3.4.2 Explicit data specification

Any class that implements the methods defined in DataSinglePassBase may be used as the data model. This includes models such as data/mg_sample_data.py that are just hard-coded arrays. Robert has used files like 'rhuva_data0.py' which read data from NetCDF files, but do not offer configurability.

3.5 Generator models

3.5.1 State variable / state handle in TxMulti framework

Section 2.1.5 describes how the TxMultiMasterSimple dispatch and multi-period loops operate.

The TxMulti system works on the concept of a state variable for each generator model, which is updated for each period in turn. The state variable holds the current period, the list of currently installed capacity and a history of all previously decommissioned capacity. The state variable is passed to the master as a 'state_handle' which is a copy of the startup state. A copy is provided as the cost function may be called from multiple threads when there is multiprocessing, and so the state cannot be kept in the model's 'self' variable. Note that a generator is free to add whatever extra information it needs to the state variable.

As the master steps through each time period in turn, each generator is given its params for that time period. The generator is requested to update the state variable and calculate the cost and output for that period.

3.5.2 get_param_count / get_param_starts

get_param_count is the method, in each generator, that tells the master how many optimisable params it would like to be allocated per time period. When constructing the simulation the master asks each generator for its param count and computes the total required. The master splits up the vector from the optimiser to give to each generator.

For TxMulti compatible multi-site generators the param_count is typically the same as the length of the params_to_site_data list (as described in section 2.1.6). If this is not set, or the generator is the SinglePassGenerator type, it will default to the number of sites that timeseries data is loaded for.

For TxMulti compatible single-site generators the param_count would typically be 1, to request optimisation of new capacity each period, or it could be 0 if the model was for a generator that had a startup state but no further installations. There is an example of this in thermal/txmultiinstantthermal.py. The TxMultiInstantFixedThermal class simply inherits from TxMultiInstantOptimisableThermal, but requests no parameters.

Generators may implement configuration parameters (typically start_min_param and start_max_param) to limit the initialisation values of the genes on the first iteration. When constructing the simulatinon the master accumulates these from each generator and passes the full list to the algorithm.

3.5.3 Using a list of new capacity instead of optimisation params

For the GE demo (web) simulation, the TxMultiGeneratorMultiSite method update_state_new_period_list is used to process directly a list of the new capacity, instead of interpreting params from the genetic algorithm. Refer to the master in getxmultimaster.py for details.

3.5.4 Base classes

3.5.4.1 SinglePassGeneratorBase

Generators based on this class are compatible with SimpleMureilMaster, which is documented in-file. Further discussion is not made here of these generators to prevent confusion with the much more versatile TxMulti variety.

3.5.4.2 TxMultiGeneratorBase

TxMultiGeneratorBase is extensively documented in-file. It lists the methods that a model needs to implement or inherit to work in the TxMulti system, but does not implement much actual functionality. The idea is that a generator can override some or all of the methods, with the methods calculate_time_period_simple and calculate_time_period_full pulling it all together.

Models that represent actual generators, where 'capacity' makes sense, will probably most easily be built as a subclass of TxMultiGeneratorMultiSite, while those that are not, such as a demand model or a missed-supply model, are most easily built by directly overriding TxMultiGeneratorBase. The models currently implemented provide good examples for these.

All generator models should override the get_simple_desc_string and get_full_desc_string methods. These methods create a user-friendly text string from a set of results from the current calculation. This user-friendly string is saved in the output pickle file and output to the screen.

3.5.4.3 TxMultiGeneratorMultiSite

TxMultiGeneratorMultiSite contains a full implementation of a multi-period capacity-building model. The state variable holds a list of all current installed capacity, a list of all decommissioned capacity, and the current period.

A full implementation is provided for the calculate_time_period_simple and calculate_time_period_full methods. The main difference between the two is that calculate_time_period_simple interfaces on an aggregated-sites basis, where calculate_time_period_full reports matrices that don't aggregate at all. calculate_time_period_simple is the method that is called by TxMultiMasterSimple. It calls the following methods in turn:

- **update_state_new_period_params:** interpret the optimisation params for this period, building capacity as directed.
- **get_site_indices:** collect a list of the site indices for sites with active capacity
- **calculate_new_capacity_cost:** given the current state, calculate the cost of the just-installed capacity. This then calls **calculate_capital_cost_site** for each site. By default, a basic linear cost model is implemented, with an extra cost applied if the site is used for the first time.
- **calculate_outputs_and_costs:** given the current state and the supply request calculate the output from each site, and the variable costs.
- **calculate_update_decommissioning:** check the state to see which sites have capacity to decommission at the end of the current period, update the state after decommissioning, and calculate the cost incurred.

calculate_time_period_simple scales up the variable costs to the time-period scale. The in-file documentation for TxMultiGeneratorBase describes the units for all of the outputs of this method. Note that if **calculate_time_period_simple** is called with the optional full_results argument set to True, a detailed dict of results is returned. When full_results is False, only the active site indices list, the total cost and the aggregate supply timeseries are returned.

Generators will commonly override the calculate_capital_cost_site method and the calculate_outputs_and_costs method. The models currently implemented, particularly TxMultiVariableGenerator, provide good examples.

Refer to the in-file documentation for full details of this implementation.

3.5.5 Demand models

Roger has implemented a demand model for Victoria that is driven by temperature data, and which takes a large set of efficiency and demand-management configuration flags. This is in demand/txmulti_victempdemand.py.

The demand model is implemented here as a single-site generator that outputs negative supply. Currently the name 'demand' in the master's dispatch order is important to identify the existence of a demand model, and to display it correctly in plots. See 5.1.4 for how this could be done more neatly.

3.5.6 Missed-supply models

Missed supply is an important model to have at the end of the dispatch order. It will evaluate if there is any unmet demand and apply penalties that will direct the optimiser to build more capacity. There are several to choose from in missed_supply/txmultimissedsupply.py. The capped missed supply model can only be used when there is a known system demand timeseries – see section 3.5.8.2 for why.

3.5.7 Max-instant-thermal models

The model TxMultiInstantMaxThermal in txmultiinstantthermal.py is an alternative last model in the dispatch order. It will build whatever capacity is needed to always meet demand.

3.5.8 Technical considerations when implementing a generator

3.5.8.1 Handling of negative param values

Section 5.3.1.1 discusses why negative optimiser param values are useful. It is important that models interpret negative param values to mean 'don't build', instead of the nonsensical 'build negative capacity'.

3.5.8.2 Handling of 'ts_demand' timeseries

Some models may desire to know the system demand timeseries for input into some forecasting function, and a missed-supply model that calculates reliability based on a fraction of MWh missed needs to know the total system demand. This is ok when the demand comes from the ts_demand data series (discussed in section 2.1.5), but not so easy when demand comes from a demand model. The issue is that the master would need to take the dynamic demand timeseries from the demand model and update any models that require it, in contrast to the once-off passing to the model of the ts_demand data series from the data model. Code to do this could be written – but without it, it's best not to require ts_demand as an input to your model.

3.5.9 Summary of generator models implemented

A list of the current generator models, a brief description of their function and a list of their configuration parameters may be obtained by running the get_config_spec_help.py script. (See section 2.2.3). For each generator model the output text will say if it implements SinglePassGenerator or TxMultiGeneratorBase. Choose only generators that are compatible with the master you are using.

3.6 Transmission models

A very simple transmission cost model is implemented in transmission/distancetxmodel.py. This simply takes the list of active site indices and builds a transmission line to the nearest trunk node. This code is incomplete – see section 5.1.3. This transmission model does not do any calculations of power flow, as these are done (well, they are aggregated as copper-plate) by the calculate_time_period_simple method of the generator models, and then further summed by TxMultiMasterSimple.

A more complex transmission model would take the per-site cost and output data from the calculate_time_period_full method of the generator models and work out the sums itself, and possibly iterate around the generator models to come to an equilibrium point. A different master would be required to do this. Section 5.2.5 discusses this in more detail.

3.7 Output format from TxMultiMasterSimple

The TxMultiMasterSimple builds up a detailed results structure at the end of a run. This is saved as a pickle file, processed into text output and optionally plotted by the function output_results. The results structure is returned to the caller – typically to runmureil.py.

The format of the results structure is a nested Python dict, with keys as follows:

- **config:** The configuration parameters for each of the models in the simulation. The keys are the section names from the configuration file(s).
- **ts_demand:** The timeseries of system demand, with keys of period as integer e.g. 2010.
- **best_params:** The final best gene.
- **opt_data:** A history of the best gene at each iteration. Format is a list of tuples of (best_gene, gene_cost, iteration_number)
- **best_results:** For the best gene, details of the state of each generator at each time period, plus totals, plus terminal values. Contains lots of detail. Run the multi-period example sim in asst5_config_multi.txt and open up the pickle file it creates (asst5.pkl) to examine them. See section 6.4.2 for help in doing this.

3.8 Multiprocessing

Multiprocessing is implemented by the genetic algorithm code. The 'gene_test' function that the genetic algorithm calls to evaluate its vectors is called in parallel on several genes at once, as the genetic algorithm evaluates multiple genes (typically tens or hundreds) on each iteration. The 'gene_test' function is defined by the simulation master as its calc_cost function, and passed to the algorithm code at configuration time.

This calc_cost function must be thread-safe so that concurrently executed calculations don't interfere with each other. This is achieved by requiring (requiring as in the code comments note it, not that the system detects it in any way) that the 'self' (the internal state) of each of the models is not changed by the calculation. This is why the TxMulti system passes around the 'state_handle' variables – a new copy is created for each cost calculation, so they do not interfere.

Multiprocessing needs to be cleaned up at the end of a run. The 'finalise' method of the master, and in turn the genetic algorithm, gives the opportunity to shut down the child processes neatly.

Multiprocessing does not currently work on Windows machines with this code. Apparently it is possible in Python, but the easy multiprocessing libraries used in the genetic algorithm code don't work so easily with Windows. The code will detect that a Windows machine is used and disable the multiprocessing.

3.9 Exceptions

The module tools/mureilexception.py defines some exception types for this project. The ConfigException exception is raised in numerous places in the code – for example from data/ncdata.py:

```
if not (len(ts_nan) == self.ts_length):
    msg = ('Data series ' + ts_name +
        ' is length {:d}, not matching {:d} of '.format(
        len(ts_nan), self.ts_length) + all_ts[0])
    raise mureilexception.ConfigException(msg, {})
```

and:

```
try:
    f = nc.NetCDFFile(infile)
except:
    msg = ('File ' + infile + ' for data series ' + series_name +
        ' was not opened.')
    raise mureilexception.ConfigException(msg, {})
```

The message is reported into the logfile, and the run will halt.

3.10 Unit and system regression testing

3.10.1 Rationale and good software practice – unit testing

Unit testing is a very effective way to ensure that your just-written module is fully functioning and ready for integration with other modules.

What you do is (and if you're doing this strictly by the rules, you'll write the unit tests before you write the code) that you prepare a set of tests that cover all of the functionality of the module, covering typical operation and also the error handling. It's often faster to prepare a unit test in a file, where you can assemble a complex configuration structure, read in data from other sources (e.g. a spreadsheet) and do a series of comparisons, than it is to poke around in an interactive Python session. As a (very big) bonus, you get to re-run the tests with no effort any time you want after that, and you can demonstrate to potential users of your code that you know it works, and that they can verify that for themselves. The test cases should be considered as a part of the codebase, so stored in the same SVN repository, and checked out with the actual functional code.

There is another benefit from writing good unit tests – it's easy for someone who can't work out what the code looks like it does to look at the unit tests and see what it does in response to various stimuli. This helps them understand what it does, and to know if the question they have about the code has already been tested, or was a 'feature' (probably a bug!) that was missed.

If you are really serious about doing good testing, it's advisable to have someone other than the code's author do some of it. One approach is to have the author do what is called 'white-box' testing – where they can see into the code, so can test that all the different paths in the code are tested, all the errors checked etc. Then a second person does the 'black-box' testing – where they are given the documentation for the code (such as the docstrings), so they know what the code should do from a functional perspective, but have little idea how it's actually done. This person will generally be very helpful in finding specification errors and in clearing up misunderstandings about what the code is supposed to do, as well as testing cases that the author thought were too obvious to test. This approach hasn't been taken to date on this project, but it would deliver additional confidence when releasing the code to the world.

As code changes it may be necessary to fix or update the unit tests so they remain current.

3.10.2 Rationale and good software practice – system regression testing

System regression testing uses fully configured simulations, as a user would do, and checks that the results haven't changed from a 'golden' run. Preferably this 'golden' run will have been thoroughly checked for correctness by the person building the test. It is vastly more difficult to test for correctness at a the system level of integration, so unit tests are important to identify any issues within modules first. A system test will typically identify issues in how the modules work together, and will occasionally pick up other bugs that the unit tests missed.

The 'regression' aspect of the testing is to build up a set of tests (unit tests plus system tests) that can be run on the current codebase and be expected to pass. This will identify if any change has been made to the code that has inadvertently broken something else. This can be done very effectively by linking it to each SVN commit, therefore identifying the culprit – or more simply by running it at the end of each day. Each user can run the test suite before they commit, so preventing issues from arising.

3.10.3 Unit tests

The subdirectories test_* contain unit test scripts for the corresponding non-test directory. The test scripts are written using the Python unittest framework. See the unittest section in the Python documentation.

For example, in test_hydro run:

```
> python test_basicpumpedhydro.py -v
```

and expect to see 'OK' printed at the end of it.

To construct a new unit test, copy an existing file, such as test_thermal/test_txmultislowthermal.py, which implements some nice tests, with data and expected results read from a spreadsheet file.

The important parts of the file that you must keep are:

At the top:

```
import sys
sys.path.append('..')

import os
import unittest
import numpy

from tools import mureilexception, testutilities
```

and then to specify a test class, here called 'TestCalcs' (the name doesn't matter), with the paths correctly set up so that the test can be run from within the test directory or from the top level as the automated tester does:

```
class TestCalcs(unittest.TestCase):
    def setUp(self):
        testutilities.unittest_path_setup(self, __file__)

    def tearDown(self):
        os.chdir(self.cwd)
```

and implement each test however you like:

```
    def test_convert_single_1(self):
    ......
```

then at the end of the file:

```
  if __name__ == '__main__':
    unittest.main()
```

Within each test, you can do whatever Python code you like. When you want to test something, use one of the following test statements (there are others in the unittest documentation too). If the test fails, the unittest framework will report where the test failed and why. Note the 'self' at the start of the assert statement. This ties the assertion to the unittest.TestCase object you are running as the test case.

To test for equality of two values (including dicts), or to evaluate any expression that gives a true/false results (from test_tools/test_global_config.py):

```
self.assertTrue((exp_pre == gc.get_config()))
```

Note that if you do this on a dict that includes numpy arrays, or if you try to compare two numpy arrays in this way, you will get a message about truth types being ambiguous across the array or something similar. To compare two numpy arrays, you can do (from test_data/test_ncdata.py):

```
self.assertTrue(numpy.allclose(exp_ts_vector, results['ts_vector']))
```

Alternatively, or if you want to compare Python lists: (from test_thermal\test_txmultislowthermal.py):

```
self.assertListEqual(out_supply.tolist(), exp_ts.tolist())
```

To test that an exception is raised, and that the message is right (from test_tools/test_global_config.py):

```
with self.assertRaises(mureilexception.ConfigException) as cm:
    gc.post_data_global_calcs()

self.assertEqual(cm.exception.msg,
    'Global calculations of time_scale_up_mult require the data_ts_length parameter to be set')
```

If you end up where you didn't want to – for example an undesired exception was raised (from test_data/test_ncdata.py):

```
self.assertEqual(False, True)
```

3.10.4 System regression tests

The system regression tests also run with the Python unittest framework. The script single_test.py in the test_regression directory calls runmureil on the test directory. Within each test directory, the script test.py calls single_test.py to do the hard work. single_test.py runs the test and then compares the output pickle file (saved as test_out.pkl) with the saved output pickle file (named in test.py). It compares the best gene at each iteration and the total cost. Note that there may be an issue of floating-point accuracy with some tests. single_test.py does round off all the cost values before doing the comparison to try to reduce the problem.

For example, in test_regression/mg_test1 run:

```
> python test.py -v
```

To set up your test, make a new subdirectory under test_regression and copy into it a test.py and __init__.py from another test, such as mg_test1. Modify the two filenames in test.py for 'config' and 'pickle', and provide a corresponding configuration file and expected test output pickle. Commit all of these files to SVN.

3.10.5 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 (section 5.1.9).

4 Web interface version (GE demo)

4.1 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.

4.2 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.

4.3 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.

5 Further work

5.1 Unfinished things

5.1.1 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.

5.1.2 Handling of negative params in gene

Section 5.3.1.1 discusses the value of having negative gene values. 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.

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 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.

5.1.7 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.

5.1.8 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.

5.1.9 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
```

5.1.10 Cleanup of SVN branches

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

5.1.11 Copyright messages

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

5.2 Next steps

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.2.4 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.

5.2.5 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, for experimenting with a full-timeseries iterative approach to determine a spot-market equilibrum, or at least provide some approximation to price signals. A full spot-market model that calculates each timestep individually would require a rework of the framework.

See the docstring for recalculate_time_period_full to see how to correctly call the methods in turn.

5.2.6 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}
```

In theory, the choice of dispatch order could also be optimised, by somehow parametrising the dispatch order decision into a list of integers and adding that to the optimiser's params vector.

5.2.7 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.

5.2.8 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.

5.2.9 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.

5.2.10 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.

5.3 Ideas for performance improvement

5.3.1 Genetic algorithm optimisations

5.3.1.1 Selection of starting points and use of negatives

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, so instead of [0, 10000] for the gene value range, it could be [-10000, 10000] or even [-20000, 10000].

Another suggestion is that for models that have start_min_param and start_max_param options, these be both set to 0, to start off with no installations at all. This puts lots of zeros in the gene pool so that empty sites come up frequently.

5.3.1.2 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.

5.3.1.3 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.

5.3.1.4 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.

5.3.2 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.

5.4 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.

5.4.1 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 regression tests where half-hourly and/or two-hourly data is used.

5.4.2 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.

5.4.3 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.

5.4.4 Unit testing of all of the models

The unit testing is pretty patchy. It would be a good task for a software student to go through each model (and other helper-functions as well) and prepare unit tests. There are plenty of examples in the test_* directories.

6 Python / System tips

6.1 Performance Improvement

6.1.1 Numeric data types

When working with numpy arrays, it's important that any floating point data is in the machine's native floating point format. This is typically float64. The data model in data/ncdata.py ensures that all floating point data is of this type, and if not, will convert it. A slow-down of 2 or more times is observed when float32 data is used.

6.1.2 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.

6.1.3 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.

When implementing models, it's important to identify opportunities for using the fast maths in numpy. The variable-generator models use numpy functions to multiply and sum along the timeseries length. For some generators, such as the pumped-hydro models and the slow-thermal, each timestep depends on the previous, and it can't be expressed in matrix-like numpy operations, so a Python loop is used. This is noticeably (up to hundreds of times) slower. If there is any opportunity to reformulate the calculation to use matrix or vector operations it is worth doing.

6.2 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.

6.3 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.

6.4 Random Python tips

6.4.1 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.

6.4.2 Working with a Pickle file

Pickle files are a nice Python thing that takes a complex structure and saves it to a file. The simulation results are saved in a pickle file. To load it in: (at a Python prompt):

```
>>> import pickle
>>> results = pickle.load(open('filename.pkl', 'rb'))
```

To work out what they keys are in the dict:

```
>>> results.keys()
```

Then to move into the dict, assuming there's a key 'totals', and continue navigating through it by using keys():

```
>>> x = results['totals']
```

Or if results.keys() doesn't work, perhaps you have a list or a tuple:

```
>>> len(x)
```

and if this is greater than 1, you can navigate with indices e.g.

```
>>> z = x[0]
```

7 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.