GOSM: Grid Operations Scenario Maker
User Manual

April 19, 2019

1 Introduction

This document explains the proper usage of the programs associated with scenario creation software in the Python package `prescient_gosm`. The acronym *GOSM* stands for *Grid Operations Scenario Maker*. *GOSM* consists of various scripts for the computation and evaluation of (generally probabilistic) scenarios related to the forecasting of renewable energy production. These scripts include `preprocessor.py`, `scenario_creator.py`, `populator.py`, and `horse_racer.py`. The `preprocessor.py` script prepares input for the scenario creation process. The `scenario_creator.py` script creates scenarios for a single day from one or more sources of uncertainty. The `populator.py` script is a barebones wrapper of the `scenario_creator.py` script, and offers the additional feature to loop over ranges of dates and produce scenarios for multiple days. Finally, the `horse_racer.py` script uses the `populator.py` script to create scenarios and then executes the simulator to produce some relevant discrimination statistics concerning different methods for creating and evaluating scenarios.

2 Installation

2.1 Installing `prescient_gosm`

You will obtain the code either from a zip file or by issuing a `git clone` command. You probably also need to get statdist and run setup.py for it.

Upon downloading the code, you will find the all scripts mentioned in this document in the directory `<Install Dir>/prescient_gosm` which will be hereafter referred to as `$PRESCIENT`.

You will need to install a collection of other programs to ensure that the scripts will run properly. These programs include *python* of course, as well as a collection of other python modules (mainly *pyomo* and the modules in the *scipy* stack). In addition to *python* modules, you must also download an optimizer (one of *CPLEX*, *Gurobi*, *IPOPT*, or another) in order to evaluate the scenarios. Informatrion about installation of these programs follows.

2.2 Python 3.4 or Later and Associated Modules

2.2.1 With Anaconda

A convenient and easy way to acquire Python 3 and the other required modules is through the Anaconda Python distribution. It may be downloaded from the following website:

`https://store.continuum.io/cshop/anaconda/`

Make sure to select the graphical installer for Python 3 for Windows, Mac OSX or Linux 32- or 64-bit, depending on your operating system. Run the executable installer to install Anaconda. This should install the relevant scientific computation libraries, *numpy*, *scipy*, and *matplotlib* as well as a collection of other modules.

2.2.2 Without Anaconda

If you do not wish to download Anaconda, you can obtain the relevant Python modules in the following manner. On UNIX and MAC OS X systems, Python 3 is usually already installed as python or python3. You can easily check this by executing the command `python` or the command `python3` in the terminal. This should start the Python interactive shell. The version number should be printed immediately after running the command. You can then exit by typing `quit()`. If executing `python` starts up python2.x whereas executing `python3` starts up python3.x, navigate to your home directory and open `.bashrc` in an editor and add `alias python=python3`. This way python3.x will be called by the `python` command line.

If you do not have the appropriate version of python installed, then you can download an installer from the website:

`https://www.python.org`

Simply download the latest version of Python and follow the instructions on the installer.

Additionally, *GOSM* requires the numpy, scipy, matplotlib, pandas, PyUtilib, networkx, and nose Python modules. For any module which is not installed, you can execute the respective command below to install the module.

- numpy: pip install numpy
- scipy:pip install scipy
- matplotlib: pip install matplotlib
- pandas: pip install pandas
- networkx: pip install networkx
- PyUtilib: pip install PyUtilib
- nose: pip install nose

2.3 Pyomo

To install Pyomo, simply open a terminal and enter `pip install pyomo`. This requires administrator access. If you do not have administrator access, the command `pip install --user pyomo` will install pyomo in the user's home directory, but it will be installed using the system python.

2.4 Optimizers

There are a couple of options for optimizers which are compatible with *prescient*. These include *CPLEX*, *Gurobi*, and *IPOPT*. The installation of any one of these and potentially other optimizers should enable the usage of the program to optimize specific problems.

2.4.1 CPLEX

Information for CPLEX can be found at the following website:

`https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

Installation of the product will likely entail creating an account on the website and may require purchasing a copy for larger optimization problems.

2.4.2 Gurobi

Information for *Gurobi* can be found on the following website:

`http://www.gurobi.com/`

2.4.3 IPOPT

If you are using a UNIX or MAC operating system, you can install IPOPT using the following commands in the terminal. Before you run these commands, check what the latest version of IPOPT is on the following website (scroll down to the newest version):

`http://www.coin-or.org/download/source/Ipopt/`

In the commands we will refer to the version of IPOPT as 3.x.x. You should change them to reflect the current version.

```
wget https://www.coin-or.org/download/source/Ipopt/Ipopt-3.x.x.tgz
mkdir Ipopt
mv Ipopt-3.x.x.tgz Ipopt/
cd Ipopt/
tar xvfz Ipopt-3.x.x.tgz
cd Ipopt-3.x.x
cd ThirdParty/Blas/
./get.Blas
cd ../Lapack
./get.Lapack
cd ../ASL
./get.ASL
cd ../Metis
./get.Metis
cd ../Mumps
```

```
./get.Mumps
cd
cd Ipopt/Ipopt-3.x.x
mkdir build
cd build
../configure
make
make test
make install
```

After running these commands, open the file `.bashrc` (`.bash_profile` on Macs) in your home directory and add the following line to the end of the document with the appropriate version number replacing the `x`'s:

```
export PATH=~/Ipopt/Ipopt-3.x.x/build/bin:$PATH
```

OR for Windows

```
set PATH=C:\path\to\Ipopt\Ipopt-3.x.x\build\bin;%PATH%
```

For more information on installing IPOPT, see the following website:

`http://www.coin-or.org/Ipopt/documentation/node10.html`

If you are running a Windows operating system, then you can download an executable from the following website:

`http://apmonitor.com/wiki/index.php/Main/DownloadIpopt`

After downloading the executable, navigate to the Downloads folder, unzip the downloaded file, and move the folder `ipopt_ampl` to the C:\ directory. Then add this folder to the PATH environment variable. You can do this by opening the Control Panel, navigating to System and Security, then to System, and finally clicking Advanced system settings. Click the button labeled *Environment Variables...* and then find the PATH variable in the User Variables and click *Edit...*. Then click *New* and write in `C:\ipopt_ampl`. If there is no PATH variable, click *New...* and give the variable the name PATH and the value `C:\ipopt_ampl`.

2.5 Setting up `GOSM`

After you have successfully installed all of the above programs, you can then install the `gosm` package. To do this, navigate to `$PRESCIENT` and run the command:

```
python setup.py install
```

This will install all the remaining programs that you will need to run the scripts in this document.

3 QuickStart

To run the prescient software you must run the following command where runner.py is a python script from the software and options.txt is an options file

```
runner.py options.txt
```

3.1 Options Files

Options files are text files which enable the user to actually execute a given script. The way an options file is structured is by listing first the program the options file is associated with on the first line. This is specified by first writing `command/exec` followed by a whitespace and the name of the *python* < *script* > it is associated with. Then each consecutive line should contain options which are to be passed to the script using typical command line syntax. The listed options are not necessarily all options needed to run the script and to view all possible options you can add the line `--help` to the < *script* > file and run the command:

```
<script> --help
```

A template and example are provided below.

3.2 Template

Every option file for whatever script you are running will be based on this template

```
command/exec <script>

# <script> is the python script you want to run

-- <option 1> <value 1>
-- <option 2> <value 2>
...
-- <option n> <value n>

# list of required and extra options followed by their set values
```

Figure 1: Template for an options file

3.3 Example

For the purposes of testing *gosm*, there exists a collection of scripts in the `$PRESCIENT/examples` directory which demonstrate basic usage of the program as well as ways in which a user may modify the behavior of the program to suit his or her needs. We look at the file *BPA_populator_example.txt* for the following example.

This file runs the populator and is reproduced in Figure 2 with its corresponding scenario creator options file in Figure 3. This program will produce scenarios for wind power generation. This script can be run when in the `$PRESCIENT/examples` directory with the command `runner.py gosm_test/BPA_populator_example.txt`.

```
command/exec populator.py

--start-date 2013-01-01
--end-date 2013-01-03

--load-scaling-factor 0.045

--output-directory gosm_test/bpa_output
--scenario-creator-options-file gosm_test/BPA_scenario_creator_example.txt
--sources-file gosm_test/bpa_sourcelist.txt
--allow-multiprocessing 0

--traceback
```

Figure 2: BPA_populator_example.txt

Following the template in Figure 1 < *script* > is just the name of the python script `populator.py`. The essential < *option* > variables for populator scripts are `--start-date` and `--end-date`, which specify start and end date (in YYYY-MM-DD format) for the range on which you wish to create scenarios, `--output-directory` which specifies where you wish to store the created scenarios, and `--scenario-creator-options-file` which specifies the file containing options specific to the creation of scenarios. The sources file must also be specifed with the `--sources-file` option. If these are not specified, the script will not execute.

The other < *option* > values are ways users can modify the execution of the populator. In this script, we see the usage of `--traceback` which prints errors in the event the program crashes. We also have `--allow-multiprocessing` and `--load-scaling-factor`. If you want to see all possible < *option* > values for a certain < *script* >, in this case for `populator.py`, you would just run the command :
`populator.py -h`

While the populator script acts more at a macroscopic level by specifying details beyond the scope of how scenarios are generated, the user specifies details of scenario construction on a day-by-day basis. The scenario creator options file in Figure 3 demonstrates this fact. We first see that there are certain essential < *option* > values which are needed to are needed to execute the scenario creator.

The most important ones for this version of *gosm* are `--use-markov-chains` and `--copula-random-walk` since the scenario generation process uses a markov chain random walk to create scenarios. These options do not need values since setting them in the options file will set those options to true. Other required < *option* > values include `--tree-template-file`, `--scenario-template-file`, `--reference-model-file` which designate the appropriate structure files which help create the output data files. If you were just running *scenario_creator.py* you would also need to provide values for `--sources-files`, `--output-directory`, and `--scenario-day` which are just like for the options for the populator options file. However, since we are running this scenario creator options file through the populator options file the values in the latter file will be set for the options.

```
command/exec scenario_creator.py

--use-markov-chains
--copula-random-walk
--planning-period-length 10H

# Options regarding file in- and output:
--sources-file gosm_test/bpa_sourcelist.txt
--output-directory gosm_test/output_scenario_creator
--scenario-template-file gosm_test/simple_nostorage_skeleton.dat
--tree-template-file gosm_test/TreeTemplate.dat
--reference-model-file ../models/knueven/ReferenceModel.py
--number-scenarios 10

# General options:
--scenario-day 2015-06-30
--wind-frac-nondispatch=0.50
```

Figure 3: BPA_scenario_creator_example.txt

Other useful < *option* > values which are not required include `--planning-period-length` which specifies the time period for the scenarios and the `--number-scenarios` which specifies the number of scenarios needed for each date. Again to see all options available run: `scenario_creator.py -h`

3.3.1 Output

After executing the script, the user should find a collection of files contained in the `$PRESCIENT/examples/gosm_test/bpa_output` directory. Within the directory `pyspdir_twostage`, the user may find a directory for each day of scenario creation. Within this file will be a collection of structured files with the extension `.dat` which are used for the purposes of simulation. These files will not be explained here. The raw scenario data will be found in the scenarios.csv file. In the plots directory, there will be a plot of the scenarios themselves as well as of the distributions used to construct the scenarios. The plot of the scenarios is reproduced here in Figure 4.

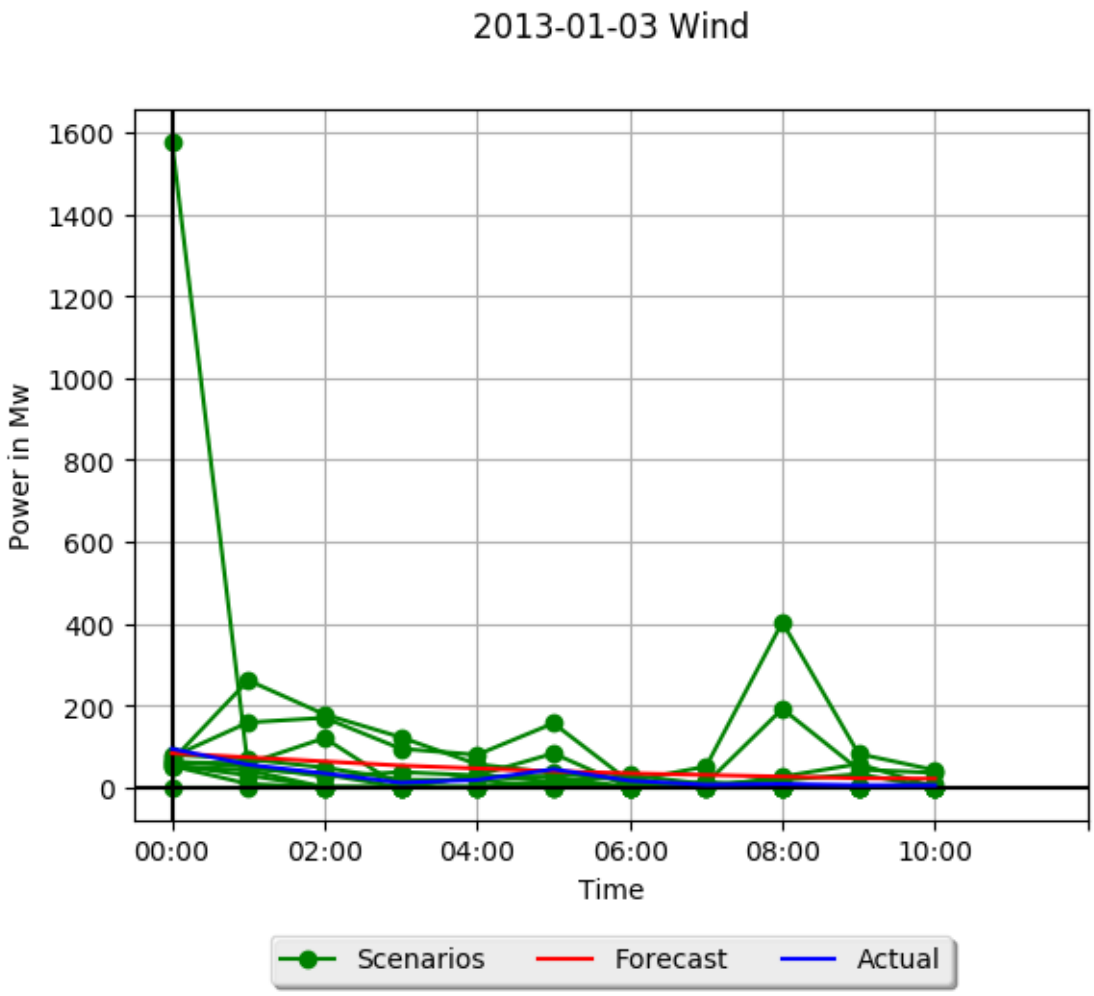

Figure 4: Scenario Plot Produced by `BPA_populator_example.txt`

4 Formatting Input Files

With all of the required program dependencies installed, you will now be able to execute various *gosm* scripts to produce and evaluation scenarios. All scripts use generally the same input formats, so it would be useful to discuss the proper formatting for input files. To use any of the scripts, there are six different types of input files:

- options files
- data files
- sources files
- segmentation files
- structure files

- capacity files

Data files are files which contain the raw forecast and actual data for various energy sources as well as the demand for power for a certain set of datetimes. Sources files are files which specify metadata about the data sources as well as auxiliary information on how to handle the sources. Segmentation files specify criteria on how to select data which is relevant for a specific datetime being analyzed. Structure files define the basic structures that are used to create scenarios. Upper bounds files specify an upper threshold to truncate generated scenarios with. Each of these are explained in further detail in the following sections. Options files specify controls for how the program as a whole runs and are described above (cf. chapter 3.1).

4.1 Data Files

Before data can be used by the various *gosm* scripts, it must be processed and formatted in a manner that the programs expect. This process currently entails creating two separate "csv" files for any time-varying data, e.g., load, wind power, or solar power. The first such file contains actual (measured) data and the other contains forecast data. Generally, the forecast data will contain quantities that were predicted day-ahead, and the actuals correspond to the realized quantities. These two files will from now on be referred to as the actuals data file and forecasts data file, respectively. In each of these files, the first column must be labeled as "datetimes" and contain datetime quantities specified in the 'YYYY-MM-DD HH:MM' format; other datetime formats may be recognized as well. The data columns must then be labeled to indicate the type of data quantity being reported – either "forecasts" or "actuals". You can also have a data file which contains both "forcasts" and "actuals" columns next to each other.

With the newest version of *gosm* we added the option for a planning period length to go through the data in increments of seconds, minutes, hours, etc. Whatever increment you would like to have for the scenario generation process, make sure the data files represent the time step. This means for instance, if your step is 5 hours your data has hourly historic data and if it is 10 minutes your data has historic data for every 10 minutes.

In general, a user will have multiple data files when modeling a single problem, each of which may contain load data, solar generation data, or wind generation data. All such data files must be formatted in the manner described above. An example snippet from a properly formatted data file is as follows:

```
datetimes,forecasts,actuals
2016-07-01 06:00,160.15, 81
2016-07-01 07:00,710.8, 680
2016-07-01 08:00,1536.78, 1905
2016-07-01 09:00,2103.23, 2113
```

Figure 5: Example file structure of a data file

4.2 Segmentation Files

For each source of uncertainty, you will need to provide a file for segmentation. Each source can of course be segmented by the same rules as well by providing the same file. The structure of the file is a series of lines where each line specifies a different criterion to segment the data by. *GOSM* operates by going through each of the lines of the file and sequentially segmenting the data by each criterion. Each line should indicate a segmentation criterion by listing in order the criterion name, column to segment by, type of segmentation, and proportion.

There are currently two types of segmentation. The first is to segment by window and this will be done if the type is set to *window*. If this method is chosen, then to compute the prediction interval for a given datetime, *GOSM* will select the proportion of points closest to the datetime in the specified field. The other method of segmentation is *enumerate*. This will select all the datetimes which match exactly in the column specified. Note that the cutpoint width is ignored in this method but it still must be provided (can be an arbitrary value). The enumeration method is especially useful if you want to use an external program to segment the data. In this case, you can just use the datetimes found by the external tool to call the segmentation method of *GOSM*.

```
# name, column_name, method, desired_proportion
forecasts,forecasts,window,0.2
enumerate,derivative_patterns,enumerate
```

Figure 6: Example of a segment criteria file

4.3 Structure Files

Structure files are dat-files including tree template files and scenario template files which specify to the program to construct PySP files for simulation with. These are more complicated structured files which can be better understood with auxiliary sources for the model by which the simulator works.

4.4 Capacity Files

If you wish to truncate the values of scenarios by a singular upper bound for certain days, you can specify how to do so with an upper bounds file. These files are structured to specify an upper bound for a date range, e.g., for the month of August, the power generated must be less than 1000 MW. To this end, the file will consist of lines in the following format:

```
01/01/00, 01/30/00, 20
```

This specifies that starting on January 1, and going on until and including January 30, the upper bound is 20. Note that dates are specified in the MM/DD/YY format. Comments are ignored and you may also specify the names of the columns as

`first_date last_date value`

before the actual rows with upper bounds, but this is not necessary.

Figure 7 shows an actual example of a capacity file.

```
# Upper bounds for wind capacities
first_date last_date value
07/01/12 03/31/13 4711
04/01/13 04/01/13 4615
04/02/13 04/30/14 4515
```

Figure 7: Example of a capacity file

4.5 Sources Files

To specify all the source-specific information for scenario generation, it is required that a sources file is written. In this file, all information for every source that is specific to that source (data files, capacities, source type) must be explicitly defined.

To specify these sources, the user must create a file with `.txt` extension and within this file, for every source, write out a Source declaration followed by a parenthesis, then the source's name followed by a comma. Then for each source parameter `key` and the corresponding value `value`, write `key="value"` separated by commas. Note each value must be surrounded by quotation marks. Then terminate the Source with a close parenthesis and a semicolon. For an example of how this might appear, see Figure 8

```
Source(name1,
key1="value1",
key2="value2"
);
Source(name2,
key1="value3",
key2="value4"
);
```

Figure 8: Example sources file

There are a couple of parameters which must be specified for each source. These are as follows

- **actuals_file**: The name of the file containing data for actual power generation values. This must have an 'actuals' column. May be the same as the forecasts file.
- **forecasts_file**: The name of the file containing data for forecasts of power generation. This must have a 'forecasts' column. May be the same as the actuals file.

- **source_type**: The type of source. This is one of 'solar', 'wind', or 'load'.
- **segmentation_file**: A file containing information about what type of segmentation will be used
- **time_step**: The time step of the actual and forecast data. The time step must be a 'H' for hours or a 'T' for minutes with an optional prefixed number.

In addition to these, there are additional optional parameters that can be specified for each source.

- **capacity_file**: The name of the file specifying daily capacities for the source.
- **is_deterministic**: Set to True if the scenario generated from this source should be simply the forecast for the source. Set to False or leave unspecified to have the scenarios generated stochastically.
- **frac_nondispatch**: The fraction of power which is nondispatchable from this source. This should be a decimal value between 0 and 1.
- **scaling_factor**: The factor by which to scale the power generation values for this source prior to scenario generation. This should be a decimal value greater than 0.
- **diurnal_pattern_file**: A name of the file specifying the diurnal pattern for a given solar source. This may be used to estimate the sunrise and sunset for this source.
- **forecasts_as_actuals**: If this options is set to True, then the actuals data will be set to the forecasts data. This option is only allowed if `is_deterministic` is also set to True.

For an example which sets some of these parameters, see Figure 9

```
Source(SoCalSolar,
        actuals_file="SP_acts.csv",
        forecasts_file="SP_forecast.csv",
        source_type="solar",
        segmentation_file="seg_solar.txt",
        capacity_file="ub.dat");

Source(NoCalSolar,
        actuals_file="NP_acts.csv",
        forecasts_file="NP_forecast.csv",
        source_type="solar",
        is_deterministic="True",
        scaling_factor="0.5",
        frac_nondispatch="0.5");

Source(load,
        actuals_file="load_history.csv",
        forecasts_file="load_history.csv",
        source_type="load");
```

Figure 9: Real example of a sources file

In addition to the above, every sources file must list one source of load data and at least one source of power (either wind or solar).

5 preprocessor.py

The preprocessor is a script to prepare the input data files for the scenario creator. It applies thresholds to the power values for each source separately. The user can set a negative and a positive threshold such that all power values greater than the positive threshold are set to this threshold and all values less than the negative threshold are also set to this threshold. This is especially useful for avoiding negative values, which may cause troubles when simulating with the created scenarios.

Running the preprocessor simply requires creating a file which lists all the files you wish to preprocess. The format of this file is composed of lines each specifying a file name and the type of source. An example of this file is in figure 10.

```
file1.txt,solar
file2.txt,wind
```

Figure 10: Example of preprocessor list

To run preprocessor, use the template for an options file (cf. chapter 3.2) and create an options file where< *script* > would be `preprocessor.py`. The important < *option* >

variables to consider are `--preprocessor-list`, which allows one to specify the file with the names of the files to preprocess, and `--output-directory` where one specifies where to store the preprocessed files. The user also will denote which types of sources to threshold and at what values with the various threshold options listed. An example of this would be `--wind-power-pos-threshold` and `--wind-power-neg-threshold`.

```
command/exec preprocessor.py

# Options regarding file in- and output:
--preprocessor-list list_of_files.txt
--output-directory output_preprocessor

# Options regarding the preprocessor:
--wind-power-pos-threshold 5000
--wind-power-neg-threshold 0
--solar-power-pos-threshold 5000
--solar-power-neg-threshold 0
--load-pos-threshold 10000
--load-neg-threshold 0
```

Figure 11: Example of an options file for preprocessor

6 scenario_creator.py

The script `scenario_creator.py` constructs scenarios for a single day of data. The exact algorithm for doing so is described in the following paragraphs.

First, the data is read from the source files. For solar sources, the average hours of sunrise and sunset are estimated for each month in order to discard all data points outside these sunshine hours.

Then the errors (differences between forecasts and actuals) are segmented by the specified segmentation criteria. Then the script calls `markov_populator.py` in order to run the Markov Chain scenario generation process.

In simplified terms the steps in the Markov Chain scenario generation process are:

- Compute a State Walk from historic data based on a description of a state specified by user arguments
- Compute a transition matrix based with probabilities weighted by the frequency with which one state follows another historically
- Compute a start state function which chooses a start state by sampling historic states which "match" the data for the date of scenario generation
- Generating random walks from a Markov chain with the start state and copulas conditioned on the state before.
- Recovering Errors from these walks and then applying them to the forecast of the day and producing scenarios which are then truncated

The scenarios can then translated into dat-files as input for a PySP-model. To do so, the user would include in the same directory as `scenario_creator.py` a scenario template file and a tree template file. The structure of these files is described in the documentation of *daps*.

We describe the proper usage of this script using the prior options file (cf. figure 3) as an example. Otherwise using the template for an options file (cf. chapter 3.2) , < *script* > would be `scenario_creator.py`. The essential < *option* > values include:

- **use-markov-chains**: The Markov Chain method is used for generating the scenarios. No value has to be set.
- **copula-random-walk**: Determines if copulas are used for the random walk. Can only be set, if use-markov-chains is also set. For multiple sources, this is a must. No value has to be set.
- **sources-file**: The sourcelist file.
- **output-directory**: The path which stores the output.
- **scenario-template-file**: The scenario template file used for creating the scenarios. For more information look into the User Manual for Daps.
- **tree-template-file**: The tree template file used for creating the scenarios. For more information look into the User Manual for Daps.
- **reference-model-file**: The reference model which is used for creating the scenarios. For more information look into the User Manual for Daps.
- **scenario-day**: If the scenario creator is run by itself and not by the populator, the scenarios are created for this day only.

Other useful < *option* > values include:

- **planning-period-length**: The length of time for the planning or scenario period as an integer directly following by a capital letter, where "H" stands for hours and "T" stands for minutes.
- **number-scenarios**: The number of scenarios generated for each day.
- **error-tolerance**: Used to speed up scenario generation when using three or more sources. This is more of an advanced topic and there is supportive documentation describing this option
- **wind-frac-nondispatch**: A float number.

After running the program, the output directory should contain a directory for each date with the desired scenarios in addition to one scenario where forecasts are used and one where actuals are used.

7 populator.py

The populator is a simple script which loops over `scenario_creator.py` on a specific date range. This is implemented by simply adding options `--start-date` and `--end-date` which specify the start and end dates for which you want to compute scenarios. These dates should be provided in YYYY-MM-DD format. The option `--scenario-creator-options-file` is used to specify an options file that is passed to `scenario_creator.py` each time it is called. This will come in handy in particular when using the script `horse_racer.py` (cf. chapter 8). However, those options will be overwritten by the options you explicitly declare in the populator's options file. An example of running `populator.py` is shown above in the documentation (cf chapter 3.3). The essential $<$ *option* $>$ values include:

- **start-date & end-date**: These dates determine the period for which scenarios are created. Both must be in the format of YYYY-MM-DD-HH:MM.
- **output-directory**: The path that stores the output.
- **scenario-creator-options-file**: The options file for the scenario creator.
- **sources-file**: The sourcelist file.

8 horse_racer.py

`horse_racer.py` is a simple script which links the results from the populator script to the simulator in *prescient*. For the specified sources, it constructs scenarios via the populator and then uses these scenarios in the simulation specified. It can be used to specify multiple experiments and the options are to be stored in a configurations file. The options for the populator and the simulator must be specified for each experiment, but either stage can be skipped by passing the option `--skip`. The format of the configurations file should be as follows:

```
Horse: <name>
Populator Options:
<populator options>
Simulator Options:
<simulator options>
```

This format can be repeated as many times as desired in the file. The following file is an example of a configurations file which specifies two experiments.

```
Horse: horse1

Populator Options:
--start-date 2015-03-01
--end-date 2015-03-31
--sources-file sourcelist.csv
--output-directory output_horse_racer/horse1
```

```
--scenario-creator-options-file run_scenario_creator_1.txt

Simulator Options:
--simulate-out-of-sample
--run-simulator
--model-directory .
--solver gurobi
--plot-individual-generators
--traceback
--output-sced-initial-conditions
--output-sced-demands
--output-sced-solutions
--output-ruc-initial-conditions
--output-ruc-solutions
--output-ruc-dispatches
--output-directory output_horse_racer/horse1_sim

Horse: horse2

Populator Options:
--start-date 2015-03-01
--end-date 2015-03-31
--sources-file sourcelist.csv
--output-directory output_horse_racer/horse2
--scenario-creator-options-file run_scenario_creator_2.txt

Simulator Options:
--simulate-out-of-sample
--run-simulator
--model-directory .
--solver gurobi
--plot-individual-generators
--traceback
--output-sced-initial-conditions
--output-sced-demands
--output-sced-solutions
--output-ruc-initial-conditions
--output-ruc-solutions
--output-ruc-dispatches
--output-directory output_horse_racer/horse2_sim
```

To execute `horse_racer.py`, you must pass two arguments in the command line, the first specifying the configurations file and the second specifying the name of the output file. If no output file is specified, the results are saved in `results.txt`. For example if we named our configurations file `horse_configurations.txt` and wanted to store the results in `sim_results.txt`, we would execute the command

```
python horse_racer.py horse_configurations.txt sim_results.txt
```

The final results of the simulation is a collection of stack graphs of the power usage at each day for each of the methods of simulating as well as a csv file containing a summary of the results for the simulation. A sample stack graph is shown in figure 12.

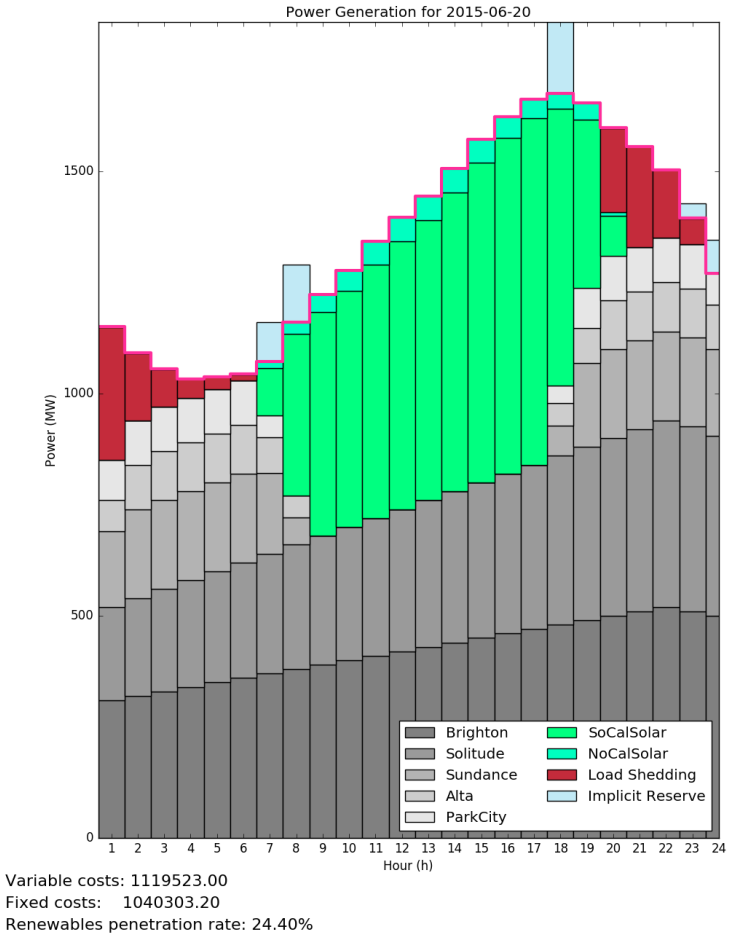

Figure 12: Example of a stack graph

The csv file produced contains information about the total cost of power generation, load shedding and over generation. The following figure shows an example of this file.

```
Horse, Total Costs, Load Shedding, Over Generation
Deterministic,14636408.3274225,2377.614206718833,239.97296887286
Stochastic,14584649.0065271,2415.0235034420048,224.16919864287
```

Figure 13: Example of a result file