GOSM: Grid Operations Scenario Maker
User Manual

May 17, 2018

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`

Once you have been granted access to the `prescient_gosm` repository, you can download the codebase with following command:

```
git clone https://github.com/jwatsonnm/prescient_gosm
```

Upon downloading the code, you will find the all scripts mentioned in this document in the directory `<Install Dir>/prescient/release/Prescient_2.0` 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. You maybe prompted to install other packages; if so, google how to install them using conda.

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

If you installed anaconda Python, you can install Pyomo using

```
conda install -c conda-forge ipopt
```

To install Pyomo with pip, 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 installed anaconda Python, you can install a version of Ipopt using

```
conda install -c conda-forge ipopt
```

Alternatively, 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
```

Note: if you have a better solver from HSL (e.g., ma57) put the source in a directory under coinhsl before the make command (e.g., put ma57.f in `ThirdParty/HSL/coinhsl/ma57`).

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 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:

- data files
- sources files
- segmentation files
- options files
- structure files
- upper bounds 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. Options files specify controls for how the program as a whole runs. 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.

3.1 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". Any additional columns in the data files will be ignored.

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
2016-07-01 06:00,160.15
2016-07-01 07:00,710.8
2016-07-01 08:00,1536.78
2016-07-01 09:00,2103.23
```

Figure 1: Example file structure of a forecasts data file

3.2 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 2: Example of a segment criteria file

```
command/exec scenario_creator.py

--sources-file sourcelist.csv
--output-directory output_scenario_creator
--hyperrectangles-file hyperrectangle_names_1source.dat
--dps-file test_dps.dat
--scenario-template-file scenario_template.dat
--tree-template-file tree_template.dat
--reference-model-file ReferenceModel.py
--scenario-day 2015-06-30
```

Figure 3: Example of an options file for scenario_creator.py

3.3 Structure Files

Structure files are dat-files that define the basic structure of hyperrectangles, day part separators and skeleton point paths. How these objects are used to create scenarios is explained in chapter 5.

A hyperrectangle is simply the product of n intervals that are subsets of $[0, 1]$. Here, n refers to the number of sources of uncertainty. A set of hyperrectangles is called a pattern. Every hyperrectangle you define must be an element of such a pattern. When writing the hyperrectangles file, you have to obey the following rules:

- The first nonempty line has to contain the keyword *Sources*, followed by a whitespace and the names of the sources, separated by whitespaces.
- Patterns start with the keyword *Pattern* followed by a colon, a whitespace and the name of the pattern.
- The hyperrectangles belonging to this pattern must be listed below, each hyperractangle in a new line starting with a dash, followed by the hyperrectangle's name, a whitespace and the intervals in the order of the sources as defined above, separated by whitespaces.
- An interval must be written as (x, y), where x defines the lower and y the upper bound. The bounds cannot have more than four decimal places.
- Every hyperrectangle of one pattern must be either disjoint to all other hyperrectangles of this pattern, or a proper subset of an other hyperrectangle. If a hyperrectangle contains a proper subset, the subset is subtracted from the hyperrectangle.

- The whole unit cube has to be covered by hyperrectangles. The easiest way to cover residual space is to create one hyperrectangle with every interval equal to (0,1).

Figure 4 shows an example of a hyperrectangle file.

```
Sources: SoCalSolar SoCalWind

Pattern: sunriseuni3
-sunu3low (0,1) (0,0.33)
-sunu3mid (0,1) (0.33,0.67)
-sunu3high (0,1) (0.67,1)

Pattern: wide1
-widelow (0,0.1) (0,0.1)
-widehigh (0.9,1) (0.9,1)
-wide1resid (0,1) (0,1)
```

Figure 4: Example of a hyperrectangle file

Day part separators (dps) are hours of the day (i.e. integer values greater than or equal 0 and less than or equal 23). They are defined within the same file as the skeleton point paths. The latter basically define ways of connecting skeleton points (which are represented by the hyperrectangles) at different day part separators. When writing the dps file, you have to obey the following rules:

- Each source starts with the phrase *Source:*, followed by a blank space and the source's name (if you are specifying paths for multiple sources, the source's name has to be *multiple*). Every following line before the next source declaration is considered to belong to this source.

- Day part separators must be defined for each source. The row in which the day part separators are given has to start with *dps*, followed by the dps hours (0-based) separated by whitespaces.

- The line before all paths of one particular source has to contain the phrase *Paths*. The following lines contain the name of the hyperrectangle pattern (e.g., *wide*) and the path which leads to this pattern (e.g., *widelow widemid*). All possible paths have to be covered. The empty path also has to be declared. It defines the skeleton point pattern at hour 0.

Figure 5 shows an example of a dps file.

```
Source: SoCalSolar
dps sunrise 10 15 sunset
Paths(decision, path)
meanonly
quick meanonly
diverse meanonly quicklow
diverse meanonly quickhigh
meanonly meanonly quicklow diverse1
meanonly meanonly quicklow diverse2
meanonly meanonly quicklow diverse3
meanonly meanonly quicklow diverse4
meanonly meanonly quickhigh diverse1
meanonly meanonly quickhigh diverse2
meanonly meanonly quickhigh diverse3
meanonly meanonly quickhigh diverse4

Source: multiple
dps sunrise 11 sunset
Paths(decision, path)
sunriseuni3
wide1 sunu3low
wide1 sunu3mid
wide1 sunu3high
sunriseuni3 sunu3low widelow
sunriseuni3 sunu3low widehigh
sunriseuni3 sunu3low wide1resid
sunriseuni3 sunu3mid widelow
sunriseuni3 sunu3mid widehigh
sunriseuni3 sunu3mid wide1resid
sunriseuni3 sunu3high widelow
sunriseuni3 sunu3high widehigh
sunriseuni3 sunu3high wide1resid
```

Figure 5: Example of a dps file

As you can see, in the special case of solar sources (or multiple sources that include at least one solar source) you can use the terms "sunrise" and "sunset" to define the day part separators for the respective hours. The definite hours will then be estimated by the program. In any other case, the first hour must be 0 and the last hour 23.

3.4 Upper Bounds 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 6 shows an actual example of an upper bounds 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 6: Example of an upper bounds file

3.5 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 7

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

Figure 7: 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'.

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

- **segmentation_file**: The name of the file specifying how to segment the 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 8

```
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 8: 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).

3.5.1 Old Sources File Format

For backwards-compatibility purposes, `prescient` supports an older csv-style format for specifying each of the sources. An example of this format is shown in Figure 9.

```
# This file is a list of sources of uncertainty, line format:
# source,actuals file,forecast file,type,segment file,bounds file
SoCalSolar,SP_acts.csv,SP_forecast.csv,solar,seg_solar.txt,ub.dat
NoCalSolar,NP_acts.csv,NP_forecast.csv,solar,seg_solar.txt,
load,load_history.csv,load_history.csv,load,seg_load.txt,
```

Figure 9: Example of old source file format

Each row in the file corresponds to a different uncertainty source. It should list in order a name, the filename of the historic data, the filename of the forecasts, the source type, the segmentation filename, and then the upper bounds file each separated by a comma. Note that you do not need to provide an upper bounds file for each source.

Note that using this sources file format, it is not possible to specify individual scaling factors or nondispatchable factors for sources, nor can a diurnal pattern file be specified.

3.6 Options Files

Options files are text files which enable the user to actually execute a given script. These are used in conjunction with the `runner.py` script located in the `gosm` directory. If `options.txt` is the options file for a certain script it can be run with the command

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

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. An example script for running `scenario_creator.py` is shown in Figure 3. Note that the listed options are not necessarily all options needed to run the script. To view all possible options you can either add the line `--help` to the options file and run the above command again, or you can look at the file `gosm_options.py`, which also contains the respective default values.

4 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 an options file for preprocessor.py

Figure 11 shows an example of the options file to run the preprocessor using the script `runner.py` (cf. section 3.6). The important options 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.

```
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.py

5 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. Note that these estimates are not used as day part separators (except if you are using `populator.py`).

Then the errors (differences between forecasts and actuals) are segmented by the specified segmentation criteria and a univariate epi-spline distribution is fitted to the resulting error data at each day part separator for each source. If the user wants to use copulas across sources to take into account, that the errors across sources are correlated, all these distributions at one day part separator are used to create a copula (which will be used like a multivariate distribution).

After that, the distributions are used to find representative points (vectors of error values) at each day part separator. This is done for each hyperrectangle by computing the conditional expected value of the respective distribution given the event of being inside of the interval bounds of the hyperrectangle.

At the end, each path is translated into a scenario with an associated probability. These probabilities are either computed by using copulas (to take into account that the errors across day part separators are correlated) or (assuming independence) by just multiplying the volumes of the hyperrectangles belonging to this path/scenario. 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*.

By setting the option `--sample-skeleton-points`, the method of computing the skeleton point values as described above is replaced by simply sampling from a uniform random distribution in $[0, 1]$ and applying the inverse cdf of the respective error distributions (or marginals). The option `--number-scenarios` specifies how many scenarios are to be created this way.

We describe the proper usage of this script using the prior options file (cf. figure 3) as an example use case. The five essential options are explained below:

- `--sources-file`: the file which contains the sources;
- `--output-directory`: the directory (possibly non-existent prior to running) to store the computed scenarios in;
- `--hyperrectangles-file`: the file which contains the possible hyperrectangles (cf. figure 4);
- `--dps-file`: the file which contains the day part separators and the paths (cf. figure 5);
- `--scenario-day`: the date of the day for which you want to create the scenarios.

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.

6 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. Figure 12 displays a sample options script for the populator. 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 7). However, those options will be overwritten by the options you explicitly declare in the populator's options file.

```
command/exec populator.py

--start-date 2015-06-20
--end-date 2015-06-25
--sources-file sourcelist.csv
--output-directory scenario_output
--scenario-creator-options-file run_scenario_creator.txt
```

Figure 12: Example of an options file for populator.py

The output is exactly the same as for `scenario_creator.py` only it has directories for each date. A difference for solar sources is, that the user of `scenario_creator.py` has to specify the day part separators at sunrise and sunset in the options file, whereas `populator.py` uses the monthly averages that are estimated in order to discard data outside the sunshine hours. Hence, the user of `populator.py` is not required to specify day part separators at sunrise and sunset for every day of the date range.

7 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 13.

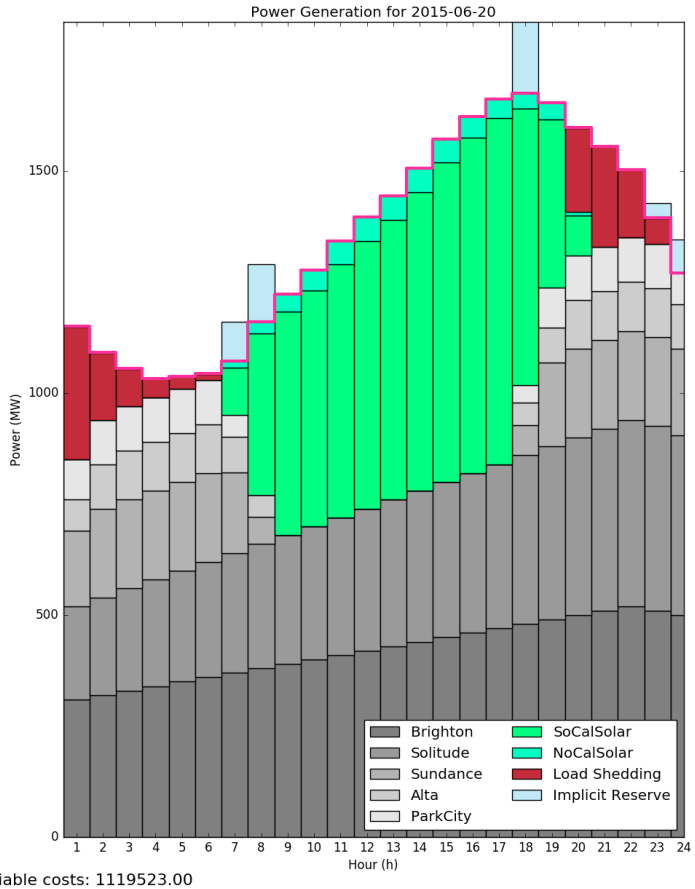

Figure 13: Example of a stack graph

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 14: Example of a result file

```
command/exec populator.py
--output-directory=simple_nostorage_scengen
--traceback
--sources-file=simple_nostorage_sources.txt
--start-date=2015-06-15
--end-date=2015-06-20
--scenario-creator-options-file=run_scenario_creator.txt
```

Figure 15: run_populator_simple_nostorage.txt

8 Example Populator Script

For the purposes of testing *gosm*, there exists a collection of scripts 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. These files are all located in the `$PRESCIENT/examples` directory and can be run from that directory with the appropriate command. For the purposes of aid in understanding the scripts, the following sections will explain one of these scripts.

8.1 run_populator_simple_nostorage.txt

This file is the simplest example of an options file which runs the populator and is reproduced in Figure 15 with its corresponding scenario creator options file in Figure 16. This program will produce scenarios for solar power generation. This script can be run when in the `$PRESCIENT/examples` directory with the command
`runner.py run_populator_simple_nostorage.txt`.

A few comments on the options specified are in order. The essential options 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 options are optional and 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.

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 16 demonstrates this fact. We first see that there are certain essential options which are those specifying the sources file, the hyperrectangles file, the dps file, and the output directory. Each of these are needed to execute the scenario creator.

In addition,the file includes the `--scenario-template-file` and `--tree-template-file` options 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. We also see options for the epi-spline which specify

```
command/exec scenario_creator.py

--sources-file sourcelist.csv
--output-directory output_scenario_creator
--hyperrectangles-file hyperrectangle_names_1source.dat
--dps-file test_dps.dat
--scenario-template-file scenario_template.dat
--tree-template-file tree_template.dat
--reference-model-file ReferenceModel.py
--scenario-day 2015-06-30
```

Figure 16: run_scenario_creator.txt

certain parameters of the specific spline model that can be varied. Then, we see there are options specifying whether to plot certain graphs which are produced during execution. The reference model must also be specified with the `--reference-model-file` option.

Since we are dealing with solar data, we see that the options `--dps-sunrise` and `--dps-sunset` are set. This is actually not essential as the program will estimate the hours of sunrise and sunset based on historic data. It would be essential if generating scenarios for a single day using the `scenario_creator`, but since our scripts run the populator, it is not required.

There are a host of other options for the `scenario_creator`. These can be seen by using the `scenario_creator.py -h` command to print the program's help page.

8.1.1 Output

After executing the script, the user should find a collection of files contained in the `$PRESCIENT/examples/simple_nostorage_scengen` 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 and will contain 24-vectors for each of the scenarios created. 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 17.

9 Advanced Topics

9.1 Spatial Copulas

If the user wishes to take into account dependencies across space when constructing scenarios, he or she will want to use spatial copulas. This can be done by specifying the `--use-spatial-copula` option and a partition file with the `--partition-file` option.

For an example of how the scenario creator options file should be structured, see Figure 18. We note that the user can specify a specific copula which can be fit with the

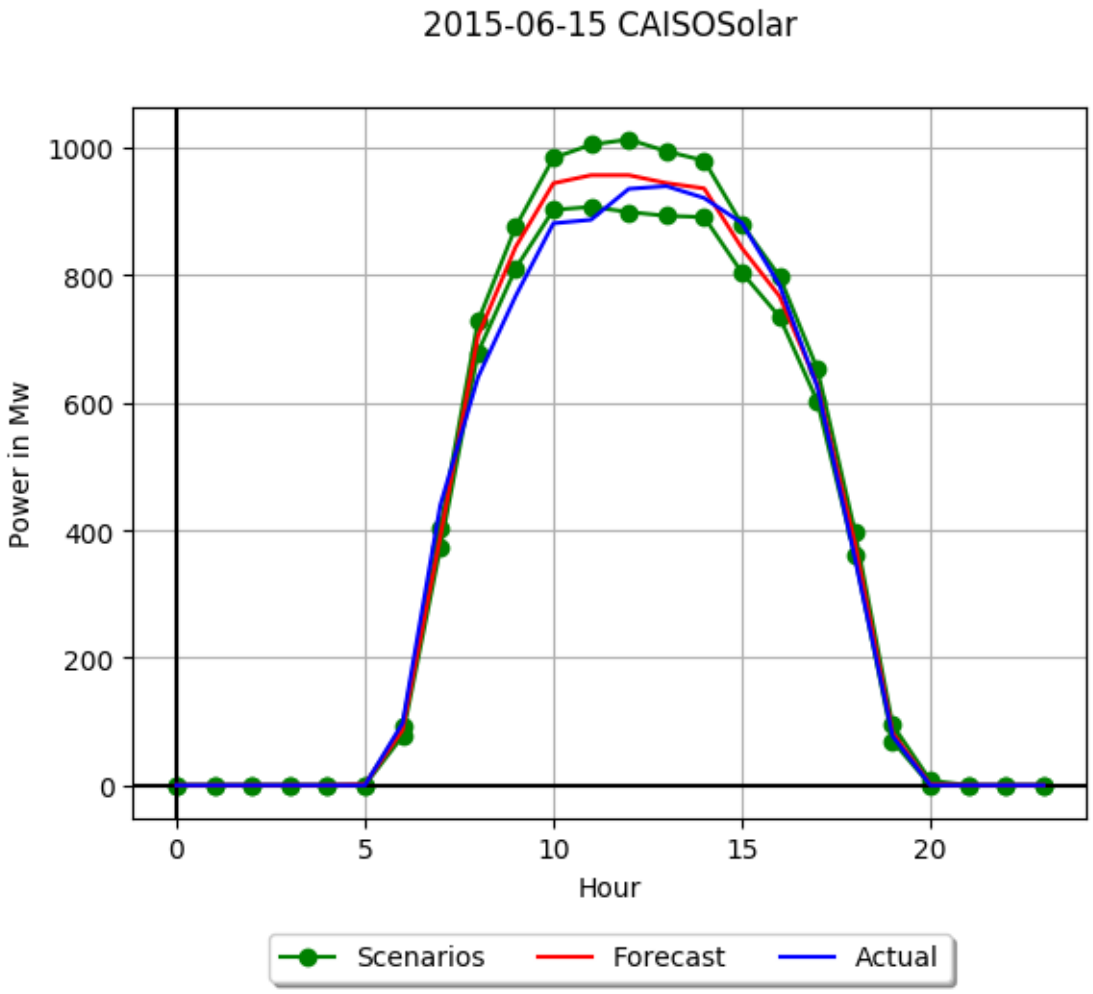

Figure 17: Scenario Plot Produced by `run_populator_simple_nostorage.txt`

`--spatial-copula` option, of which currently only the Gaussian Copula is available as an option. The user can also specify the `--use-same-paths-across-correlated-sources` which means that for a specific scenario, on each of the correlated sources, the same hyperrectangle set will be used. For uncorrelated sources, different sources may use different hyperrectangles.

9.1.1 Partition Files

For the purposes of using spatial copulas, it is necessary to specify which sources are related. For *GOSM*, this is done using a partition file. In it, sources which are related are grouped together and any singleton source is specified in its own section.

The structure of the file is as follows:

```
<Group1 Name>:
-<Source1>
-<Source2>
...
```

```
command/exec scenario_creator.py

# Options regarding file in- and output:
--sources-file gosm_test/bpa_sourcelist.csv
--output-directory gosm_test/output_scenario_creator
--hyperrectangles-file gosm_test/hyperrectangle_names_1source.dat
--dps-file gosm_test/spatial_cutpoints.txt

# Options regarding all distributions:
--plot-variable-gap 10
--plot-pdf 1
--plot-cdf 0
--cdf-inverse-tolerance 1.0e-3

--use-spatial-copula
--spatial-copula gaussian-copula

--use-same-paths-across-correlated-sources

--partition-file gosm_test/partition.txt
```

Figure 18: An example scenario creator file for spatial copulas

```
<GroupN Name>:
-<SourceN1>
-<SourceN2>
...
Singletons:
-<SingleSource1>:
-<SingleSource2>:
```

For good measure, an actual partition file example is in Figure 19.

```
# Partitions File

Group 1:
-ARW
-PSW
-WFW

Singletons:
-STL
-TRW
```

Figure 19: Example of a partition file

9.2 Disaggregated Sources

In certain circumstances, it may be the case that a user has a source which is actually an aggregation of multiple sources. This user may want to generate scenarios using this aggregate source and then have the generated scenarios be disaggregated according to certain proportions. This can be done by setting the "aggregate" option for a given source to "True" and then setting the "disaggregation_file" to a file which has the following format:

```
source,proportion
<Source1>,<Proportion1>
<Source2>,<Proportion2>
...
```

In the above, `<Source1>` refers to the name of the first component source of the aggregate source and `<Proportion1>` is the corresponding proportion of the power which the source should be producing.

An example of this disaggregation file is presented in Figure 20.

```
source,proportion
A,0.2
B,0.4
C,0.4
```

Figure 20: Example of a disaggregation file

A sources file which might appear with this disaggregation file is also shown in Figure 21.

```
Source(Wind,
        actuals_file="gosm_test/2012-2013_BPA_forecasts_actuals.csv",
        forecasts_file="gosm_test/2012-2013_BPA_forecasts_actuals.csv",
        source_type="wind",
        segmentation_file="gosm_test/segment_bpa.txt",
        capacity_file="gosm_test/manual_ub.dat",
        aggregate="True",
        disaggregation_file="gosm_test/wind_generators.txt");
Source(Load,
        actuals_file="gosm_test/CAiso-TAC_demand_12-15.csv",
        forecasts_file="gosm_test/CAiso-TAC_demand_12-15.csv",
        source_type="load",
        segmentation_file="gosm_test/no_segmentation.txt");
```

Figure 21: Example of a disaggregation file