Prescient

Release 2.2.2

Prescient Developers

Apr 04, 2023

CONTENTS:

Prescient is a python library that provides production cost modeling capabilities for power generation and distribution networks.

CHAPTER

ONE

USING PRESCIENT

1.1 Installation

The Prescient python package can be installed using pip, or it can be installed from source. Python and a linear solver are prerequisites for either installation method.

To install Prescient, follow these steps:

1.1.1 Install python

Prescient requires python 3.7 or later. We recommend installing Anaconda to manage python and other dependencies.

1.1.2 Install a linear solver

Prescient requires a mixed-integer linear programming (MILP) solver that is compatible with Pyomo. Options include open source solvers such as CBC or GLPK, and commercial solvers such as CPLEX, Gurobi, or Xpress.

The specific mechanics of installing a solver is specific to the solver and/or the platform. An easy way to install an open source solver on Linux and Mac is to install the CBC Anaconda package into the current conda environment:

```
conda install -c conda-forge coincbc
```

Tip: Be sure to activate the correct python environment before running the command above.

Binaries for Windows and other platforms may be available from https://github.com/coin-or/Cbc/releases.

Note that the CBC solver is used in most Prescient tests, so you may want to install it even if you intend to use another solver in your own runs.

1.1.3 Install Using Pip

Prescient is available as a python package that can be installed using pip. To install the latest release of Prescient use the following command:

> pip install gridx-prescient

Be sure the intended python environment is active before issuing the command above.

1.1.4 Install From Source

You may want to install from source if you want to use the latest pre-release version of the code, or if you want to modify/contribute to the code yourself. The steps required to install Prescient from source are described below:

Get Prescient source code

The latest version of Prescient can be acquired as source from the Prescient github project, either by downloading a zip file of the source code or by cloning the *main* branch of the github repository.

Install Python Dependencies

The python environment where you run Prescient must include a number of prerequisites. You may want to create a python environment specifically for Prescient. To create a new Anaconda environment and install Prescient's prerequisites into the new environment, issue the following command from the root folder of the Prescient source code:

```
conda env create -f environment.yml
```

The command above will create an environment named *prescient*. To use a different name for the environment, add the *-n* option to the command above:

```
conda env create -n nameOfYourChoice -f environment.yml
```

Once you have create the new environment, make it the active environment:

```
conda activate prescient
```

If you are using something other than Anaconda to manage your python environment, use the information in *environment.yml* to identify which packages to install.

Install Egret

When installing Prescient from the latest version of the source code, Egret may need to be installed manually because pre-release versions of Prescient sometimes depend on pre-release versions of EGRET. Install EGRET from source according to the instructions *here <https://github.com/grid-parity-exchange/Egret/blob/main/README.md>*.

Install the Prescient python package

The steps above configure a python environment with Prescient's prerequisites. Now we must install Prescient itself. From the prescient python environment, issue the following command:

```
pip install -e .
```

This will update the active python environment to include Prescient's source code. Any changes to Prescient source code will take affect each time Prescient is run.

This command will also install a few utilities that Prescient users may find useful, including *runner.py* (see *Running Prescient*).

Verify your installation

Prescient is packaged with tests to verify it has been set up correctly. To execute the tests, issue the following command:

```
pytest -v prescient/simulator/tests/test_simulator.py
```

This command runs the tests using the CBC solver and will fail if you haven't installed CBC. The tests can take as long as 30 minutes to run, depending on your machine. If Prescient was installed correctly then all tests should pass.

1.2 Running Prescient

There are three ways to launch and run Prescient:

- With a configuration file, *using runner.py*
- With command line options, *using the prescient.simulator module*
- From python code, *using in-code configuration*

In all three cases, the analyst supplies configuration values that identify input data and dictate which options to use during the Prescient simulation. Configuration options can be specified in a configuration file, on the command line, in-code, or a combination of these methods, depending on how Prescient is launched.

To see what configuration options are available, see *Configuration Options*.

1.2.1 Launch with runner.py

Prescient can be run using *runner.py*, a utility which is installed along with Prescient (see *Install Egret*). Before executing *runner.py*, you must create a configuration file indicating how Prescient should be run. Here is an example of a configuration file that can be used with *runner.py*:

```
command/exec simulator.py
--data-directory=example_scenario_input
--output-directory=example_scenario_output
--input-format=rts-gmlc
--run-sced-with-persistent-forecast-errors
--start-date=07-11-2024
--num-days=7
--sced-horizon=1
--sced-frequency-minutes=10
--ruc-horizon=36
```

Because runner.py can potentially be used for more than launching Prescient, the first line of the configuration file must match the line shown in the example above. Otherwise runner.py won't know that you intend to run Prescient.

All subsequent lines set the value of a configuration option. Configuration options are described in *Configuration Options*.

Once you have the configuration file prepared, you can launch Prescient using the following command:

```
runner.py config.txt
```

where *config.txt* should be replaced with the name of your configuration file.

1.2.2 Launch with the *prescient.simulator* module

Another way to run Prescient is to execute the *prescient.simulator* module:

```
python -m prescient.simulator <options>
```

where *options* specifies the configuration options for the run. An example might be something like this:

```
python -m prescient.simulator --data-directory=example_scenario_input --output-
↪directory=example_scenario_output --input-format=rts-gmlc --run-sced-with-persistent-
↪forecast-errors --start-date=07-11-2024 --num-days=7 --sced-horizon=1 --sced-frequency-
↪minutes=10 --ruc-horizon=36
```

Configuration options can also be specified in a configuration file:

```
python -m prescient.simulator --config-file=config.txt
```

You can combine the *–config-file* option with other command line options. The contents of the configuration file are effectively inserted into the command line at the location of the *–config-file* option. You can override values in a configuration file by repeating the option at some point after the *–config-file* option.

Running the *prescient.simulator* module allows you to run Prescient without explicitly installing it, as long as Prescient is found in the python module search path.

1.2.3 Running Prescient from python code

Prescient can be configured and launched from python code:

```
from prescient.simulator import Prescient

Prescient().simulate(
        data_path='deterministic_scenarios',
        simulate_out_of_sample=True,
        run_sced_with_persistent_forecast_errors=True,
        output_directory='deterministic_simulation_output',
        start_date='07-10-2020',
        num_days=7,
        sced_horizon=4,
        reserve_factor=0.0,
        deterministic_ruc_solver='cbc',
        sced_solver='cbc',
        sced_frequency_minutes=60,
        ruc_horizon=36,
        enforce_sced_shutdown_ramprate=True,
        no_startup_shutdown_curves=True)
```

The code example above creates an instance of the Prescient class and passes configuration options to its *simulate()* method. An alternative is to set values on a configuration object, and then run the simulation after configuration is done:

```
from prescient.simulator import Prescient

p = Prescient()

config = p.config
config.data_path='deterministic_scenarios'
config.simulate_out_of_sample=True
config.run_sced_with_persistent_forecast_errors=True
config.output_directory='deterministic_simulation_output'
config.start_date='07-10-2020'
config.num_days=7
config.sced_horizon=4
config.reserve_factor=0.0
config.deterministic_ruc_solver='cbc'
config.sced_solver='cbc'
config.sced_frequency_minutes=60
config.ruc_horizon=36
config.enforce_sced_shutdown_ramprate=True
config.no_startup_shutdown_curves=True

p.simulate()
```

A third option is to store configuration values in a *dict*, which can potentially be shared among multiple runs:

```
from prescient.simulator import Prescient

options = {
    'data_path':'deterministic_scenarios',
```

(continues on next page)

(continued from previous page)

```
    'simulate_out_of_sample':True,
    'run_sced_with_persistent_forecast_errors':True,
    'output_directory':'deterministic_simulation_output'
}

Prescient().simulate(**options)
```

These three methods can be used together quite flexibly. The example below demonstrates a combination of approaches to configuring a prescient run:

```
from prescient.simulator import Prescient

simulator = Prescient()

# Set some configuration options using the simulator's config object
config = simulator.config
config.data_path='deterministic_scenarios'
config.simulate_out_of_sample=True
config.run_sced_with_persistent_forecast_errors=True
config.output_directory='deterministic_simulation_output'

# Others will be stored in a dictionary that can
# potentially be shared among multiple prescient runs
options = {
    'start_date':'07-10-2020',
    'sced_horizon':4,
    'reserve_factor':0.0,
    'deterministic_ruc_solver':'cbc',
    'sced_solver':'cbc',
    'sced_frequency_minutes':60,
    'ruc_horizon':36,
    'enforce_sced_shutdown_ramprate':True,
    'no_startup_shutdown_curves':True,
}

# And finally, pass the dictionary to the simulate() method,
# along with an additional function argument.
simulator.simulate(**options, num_days=7)
```

1.3 Configuration Options

1.3.1 Overview

Prescient configuration options are used to indicate how the Prescient simulation should be run. Configuration options can be specified on the command line, in a text configuration file, or in code, depending on how Prescient is launched (see *Running Prescient*).

Each configuration option has a name, a data type, and a default value. The name used on the command line and the name used in code vary slightly. For example, the number of days to simulate is specified as *--num-days* on the command line, and *num_days* in code.

1.3.2 Option Data Types

Most options use self-explanatory data types like *String*, *Integer*, and *Float*, but some data types require more explanation and may be specified in code in ways that are unavailable on the command line:

Table 1: Configuration Data Types

Data type	Command-line/config file usage	In-code usage
Path	A text string that refers to a file or folder. Can be relative or absolute, and may include special characters such as ~.	Same as command-line
Date	A string that can be converted to a date, such as *1776-07-04*.	Either a string or a datetime object.
Flag	Simply include the option to set it to true. For example, the command below sets *simulate_out_of_sample* to true: `runner.py --simulate-out-of-sample`	Set the option by assigning True or False: `config.simulate_out_of_sample = True`

Module	Refer to a python module in one of the following ways:
	• The name of a python module (such as *prescient.simulator.prescient*)
	• The path to a python file (such as *prescient/simulator/prescient.py*)
		In addition to the two string options available to the command-line, code may also use a python module object. For example:
		`import my_custom_data_provider`
		`config.data_provider = my_custom_data_provider`

1.3.3 List of Configuration Options

The table below describes all available configuration options.

Table 2: Configuration Options

Command-line Option	In-Code Configuration Property	Argument	Description

--config-file	config_file	Path. Default=None.	Path to a file holding configuration options. Can be absolute or relative. Cannot be set in code directly on a configuration object, but can be passed to a configuration object's *parse_args()* function:
			`p = Prescient()`
			`p.config.parse_args(["--config-file", "my-config.txt"])`
			See *Launch with runner.py* for a description of configuration file syntax.

General Options			
--start-date	start_date	Date. Default=2020-01-01.	The start date for the simulation.
--num-days	num_days	Integer. Default=7	The number of days to simulate.
Data Options			
--data-path or --data-directory	data_path	Path. Default=input_data.	Path to a file or folder where input data is located. Whether it should be a file or a folder depends on the input format. See *Input Data*.
--input-format	input_format	String. Default=dat.	The format of the input data. Valid values are *dat* and *rts_gmlc*. Ignored when using a custom data provider. See *Input Data*.
--data-provider	data_provider	Module. Default=No custom data provider.	A python module with a custom data provider that will supply data to Prescient during the simulation. Don't specify this option unless you are using a custom data provider; use data_path and input_format instead. See *Custom Data Providers*.
--output-directory	output_directory	Path. Default=outdir.	The path to the root directory to which all generated simulation output files and associated data are written.
RUC Options			
--ruc_every-hours	ruc_every_hours	Integer. Default=24	How often a RUC is executed, in hours. Default is 24. Must be a divisor of 24.
--ruc-execution-hour	ruc_execution_hour	Integer. Default=16	Specifies an hour of the day the RUC process is executed. If multiple RUCs are executed each day (because *ruc_every_hours* is less than 24), any of the execution times may be specified. Negative values indicate hours before midnight, positive after.
--ruc-horizon	ruc_horizon	Integer. Default=48	The number of hours to include in each RUC. Must be >= *ruc_every_hours* and <= 48.

continues on next page

Table 2 – continued from previous page

Command-line Option	In-Code Configuration Property	Argument	Description
--ruc-prescience-hour	ruc_prescience_hour	Integer. Default=0.	The number of initial hours of each RUC in which linear blending of forecasts and actual values is done, making some near-term forecasts more accurate.
--run-ruc-with-next-day-data	run_ruc_with_next_day_data	Flag. Default=false.	If false (the default), never use more than 24 hours of forecast data even if the RUC horizon is longer than 24 hours. Instead, infer values beyond 24 hours. If true, use forecast data for the full RUC horizon.
--simulate-out-of-sample	simulate_out_of_sample	Flag. Default=false.	If false, use forecast input data as both forecasts and actual values; the actual value input data is ignored. If true, values for the current simulation time are taken from the actual value input, and actual values are used to blend near-term values if *ruc_prescience_hour* is non-zero.
--ruc-network-type	ruc_network_type	String. Default=ptdf.	Specifies how the network is represented in RUC models. Choices are: * ptdf – power transfer distribution factor representation * btheta – b-theta representation
--ruc-slack-type	ruc_slack_type	String. Default=every-bus.	Specifies the type of slack variables to use in the RUC model formulation. Choices are: * every-bus – slack variables at every system bus * ref-bus-and-branches – slack variables at only reference bus and each system branch
--deterministic-ruc-solver	deterministic_ruc_solver	String. Default=cbc.	The name of the solver to use for RUCs.
--deterministic-ruc-solver-options	deterministic_ruc_solver_options	String. Default=None.	Solver options applied to all RUC solves.
--ruc-mipgap	ruc_mipgap	Float. Default=0.01.	The mipgap for all deterministic RUC solves.
--output-ruc-initial-conditions	output_ruc_initial_conditions	Flag. Default=false.	Print initial conditions to stdout prior to each RUC solve.
--output-ruc-solutions	output_ruc_solutions	Flag. Default=false.	Print RUC solution to stdout after each RUC solve.
--write-deterministic-ruc-instances	write_deterministic_ruc_instances	Flag. Default=false.	Save each individual RUC model to a file. The date and time the RUC was executed is indicated in the file name.

continues on next page

Table 2 – continued from previous page

Command-line Option	In-Code Configuration Property	Argument	Description
--deterministic-ruc-solver-plugin	deterministic_ruc_solver_plugin	Module. Default=None.	If the user has an alternative method to solve RUCs, it should be specified here, e.g., my_special_plugin.py. **Note:** This option is ignored if --*simulator-plugin* is used.
SCED Options			
--sced-frequency-minutes	sced_frequency_minutes	Integer. Default=60.	How often a SCED will be run, in minutes. Must divide evenly into 60, or be a multiple of 60.
--sced-horizon	sced_horizon	Integer. Default=1	The number of time periods to include in each SCED. Must be at least 1.
--run-sced-with-persistent-forecast-errors	run_sced_with_persistent_forecast_errors	Flag. Default=false.	If true, then values in SCEDs use persistent forecast errors. If false, all values in SCEDs use actual values for all time periods, including future time periods. See *Forecast Smoothing*.
--enforce-sced-shutdown-ramprate	enforce_sced_shutdown_ramprate	Flag. Default=false.	Enforces shutdown ramp-rate constraints in the SCED. Enabling this option requires a long SCED look-ahead (at least an hour) to ensure the shutdown ramp-rate constraints can be statisfied.
--sced-network-type	sced_network_type	String. Default=ptdf.	Specifies how the network is represented in SCED models. Choices are: * ptdf – power transfer distribution factor representation * btheta – b-theta representation
--sced-slack-type	sced_slack_type	String. Default=every-bus.	Specifies the type of slack variables to use in SCED models. Choices are: * every-bus – slack variables at every system bus * ref-bus-and-branches – slack variables at only reference bus and each system branch
--sced-solver	sced_solver	String. Default=cbc.	The name of the solver to use for SCEDs.
--sced-solver-options	sced_solver_options	String. Default=None.	Solver options applied to all SCED solves.
--print-sced	print_sced	Flag. Default=false.	Print results from SCED solves to stdout.
--output-sced-initial-conditions	output_sced_initial_conditions	Flag. Default=false.	Print SCED initial conditions to stdout prior to each solve.
--output-sced-loads	output_sced_loads	Flag. Default=false.	Print SCED loads to stdout prior to each solve.
--write-sced-instances	write_sced_instances	Flag. Default=false.	Save each individual SCED model to a file. The date and time the SCED was executed is indicated in the file name.

continues on next page

Table 2 – continued from previous page

Command-line Option	In-Code Configuration Property	Argument	Description
Output Options			
--disable-stackgraphs	disable_stackgraphs	Flag. Default=false.	Disable stackgraph generation.
--output-max-decimal-places	output_max_decimal_places	Integer. Default=6.	The number of decimal places to output to summary files. Output is rounded to the specified accuracy.
--output-solver-logs	output_solver_logs	Flag. Default=false.	Whether to print solver logs to stdout during execution.
Miscellaneous Options			
--reserve-factor	reserve_factor	Float. Default=0.0.	The reserve factor, expressed as a constant fraction of demand, for spinning reserves at each time period of the simulation. Applies to both RUC and SCED models.
--no-startup-shutdown-curves	no_startup_shutdown_curves	Flag. Default=False.	If true, then do not infer startup/shutdown ramping curves when starting-up and shutting-down thermal generators.
--symbolic-solver-labels	symbolic_solver_labels	Flag. Default=False.	Whether to use symbol names derived from the model when interfacing with the solver.
--enable-quick-start-generator-commitment	enable_quick_start_generator_commitment	Flag. Default=False.	Whether to allow quick start generators to be committed if load shedding would otherwise occur.
Market and Pricing Options			
--compute-market-settlements	compute_market_settlements	Flag. Default=False.	Whether to solve a day-ahead market as well as real-time market and report the daily profit for each generator based on the computed prices.
--day-ahead-pricing	day_ahead_pricing	String. Default=aCHP.	The pricing mechanism to use for the day-ahead market. Choices are: * LMP – locational marginal price * ELMP – enhanced locational marginal price * aCHP – approximated convex hull price.
--price-threshold	price_threshold	Float. Default=10000.0.	Maximum possible value the price can take. If the price exceeds this value due to Load Mismatch, then it is set to this value.
--reserve-price-threshold	reserve_price_threshold	Float. Default=10000.0.	Maximum possible value the reserve price can take. If the reserve price exceeds this value, then it is set to this value.
Plugin Options			

continues on next page

Table 2 – continued from previous page

Command-line Option	In-Code Configuration Property	Argument	Description
--plugin	plugin	Module. Default=None.	Python plugins are analyst-provided code that Prescient calls at various points in the simulation process. See *Customizing Prescient with Plugins* for details. After Prescient has been initialized, the configuration object's *plugin* property holds plugin-specific setting values.
--simulator-plugin	simulator_plugin	Module. Default=None.	A module that implements the engine interface. Use this option to replace methods that setup and solve RUC and SCED models with custom implementations.

1.4 Input Data

1.4.1 Custom Data Providers

1.5 Results and Statistics Output

Under Construction

Documentation coming soon

1.6 Customizing Prescient with Plugins

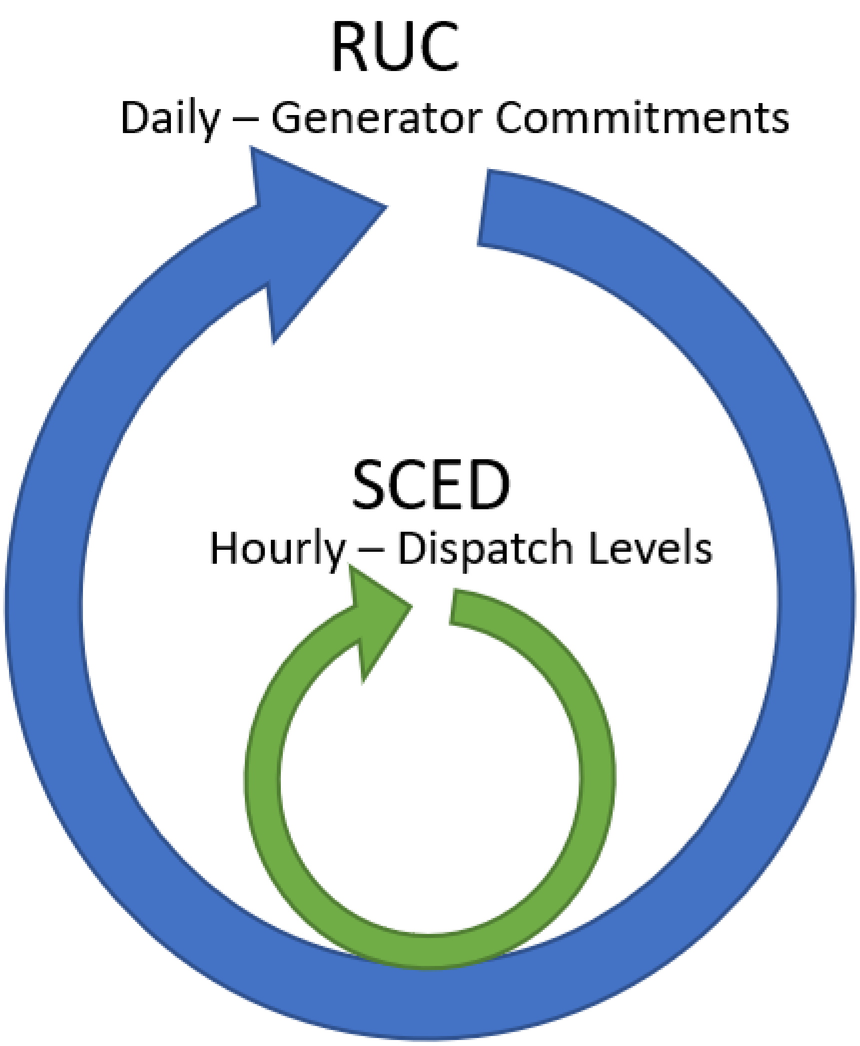
Under Construction

Documentation coming soon

CHAPTER TWO

MODELING CONCEPTS

2.1 The Prescient Simulation Cycle



Prescient simulates the operation of a power generation network throughout a study horizon, finding the set of operational choices that satisfy demand at the lowest possible cost.

A Prescient simulation consists of two repeating cycles, one nested in the other. The outer cycle is the Reliability Unit Commitment (RUC) planning cycle, which schedules changes in dispatchable generators' online status during the cycle's period. The inner, more frequent cycle is the Security Constrained Economic Dispatch (SCED) cycle, which determines dispatch levels for dispatchable generators.

2.1.1 The RUC Cycle

The RUC cycle periodically generates a RUC plan. A RUC plan consists of two types of data: a unit commitment schedule and optionally a pricing schedule (when *compute-market-settlements* is True). The unit commitment schedule indicates which dispatchable generators should be activated or deactivated during upcoming time periods. The pricing schedule sets the contract price for expected power delivery and for reserves (ancillary service products). The RUC plan reflects the least expensive way to satisfy predicted loads while honoring system constraints.

A new RUC plan is generated at regular intervals, at least once per day. A new RUC plan always goes into effect at midnight of each day. If more than one RUC plan is generated each day, then additional RUC plans take effect at equally spaced intervals. For example, if 3 RUC plans are generated each day, then one will go into effect at midnight, one at 8:00 a.m., and one at 4:00 p.m. Each RUC plan covers the time period that starts when it goes into effect and ends just as the next RUC plan becomes active.

A RUC plan is based on the current state of the system at the time the plan is generated (particularly the current dispatch and up- or down-time for dispatchable generators), and on forecasts for a number of upcoming time periods. The forecasts considered when forming a RUC plan must extend at least to the end of the RUC's planning period, but typically extend further into the future in order to avoid poor choices at the end of the plan ("end effects"). The amount of time to consider when generating a RUC plan is known as the RUC horizon. A commonly used RUC horizon is 48 hours.

The simulation can be configured to generate RUC plans some number of hours before they take effect. This is done by specifying a time of day for one of the plans to be generated. The gap between the specified generation time and the next time a RUC plan is scheduled to take effect is called the RUC gap. Each RUC plan still covers the expected time period, from the time the plan takes effect until the next RUC plan takes effect, but its decisions will be based on what is known at the time the RUC plan is generated.

2.1.2 The SCED Cycle

The SCED process selects dispatch levels for all active dispatchable generators in the current simulation time period. Dispatch levels are determined using a process that is very similar to that used to build a RUC plan. The current state of the system, together with forecasts for a number of future time periods, are examined to select dispatch levels that satisfy current loads and forecasted future loads at the lowest possible cost.

The SCED cycle is more frequent than the RUC cycle, with new dispatch levels selected at least once an hour. The SCED honors unit commitment decisions made in the RUC plan; whether each generator is committed or not is dictatated by the RUC schedule currently in effect.

Costs are also determined with each SCED, based on dispatchable generation selected by the SCED process, the commitment and start-up costs as selected by the associated RUC process, as well as current actual demands and non-dispatchable generation levels.

2.2 Time Series Data Streams

Prescient uses time series data from two data streams, the real-time stream (i.e., actuals) and the forecast stream. As their names imply, the real-time stream includes data that the simulation should treat as actual values that occur at specific times in the simulation, and the forecast stream includes forecasts for time periods that have not yet occured in the simulation.

Both streams consist of time-stamped values for loads and non-dispatchable generation data.

2.2.1 Real-Time Data (Actuals)

The real-time data stream provides data that the simulation should treat as actual values. Real-time values are typically used only when the simulation reaches the corresponding simulation time.

Real-time data can be provided at any time interval. The real-time data interval generally matches the SCED interval (see *sced-frequency-minutes*), but this is not a requirement. If the SCED interval does not match the real-time interval then real-time data will be interpolated or discarded as needed to match the SCED interval.

2.2.2 Forecasts

Forecast data are provided by the forecast data stream. The frequency of data provided through the forecast stream must be hourly.

New forecasts are retrieved each time a new RUC plan is generated. The forecasts retrieved in a given batch are those required to satisfy the RUC horizon (see *ruc-horizon*), starting with the RUC activation time.

Forecast Smoothing

As forecasts are retrieved from the forecast data stream, they may be adjusted so that near-term forecasts are more accurate than forecasts further into the future. This serves two purposes: first, to avoid large jumps in timeseries values due to inaccurate forecasts; and second, to model how forecasts become more accurate as their time approaches.

The number of forecasts to be smoothed is determined by the *ruc-prescience-hour* configuration option. Values for the current simulation time are set equal to their actual value, ignoring data read from the forecast stream. Values for `ruc-prescience-hour` hours after the current simulation time are set equal to data read from the forecast stream. Between these two times, values are a weighted average of the values provided by the actuals and forecast data streams. The weights vary linearly with where the time falls between the current time and the ruc prescience hour. For example, if `ruc-prescience-hour` is 8, then the adjusted forecast for 2 hours after the current simulation time will be `0.25*forecast + 0.75*actual`.

Note that blending weights are determined relative to the current simulation time when the RUC is generated, not relative to the time the RUC goes into effect.

Real-Time Forecast Adjustments

Forecasts are adjusted further each time a SCED is run. This is done by comparing the forecast for the current time with the actual value for the current time. The ratio of these two values is calculated, then used as a scaling factor for forecast values. For example, if the forecast for a value was 10% too high, all future forecasts for the same value are reduced by 10%.

Note: If *run-sced-with-persistent-forecast-errors* is false, then SCEDs will use actual values for all time periods. Forecasts will still be used for RUCs, but SCEDs will be based entirely on actual values, even for future time periods.

2.3 Reserves and Ancillary Services

2.4 Energy Markets and Pricing

CHAPTER THREE

EXAMPLES AND TUTORIALS

CHAPTER

FOUR

REFERENCE

4.1 Input Data

4.1.1 The CSV Input File Format

The system being modeled by Prescient is read from a set of CSV files. The CSV files and their format is based on the RTS-GMLC format. Prescient uses only a subset of the columns present in RTS-GMLC format. This document identifies the columns read by Prescient, their meaning, and how they are represented in the Egret model used by Prescient at runtime. Any additional columns be present in the input are ignored.

There are six required CSV files and two optional CSV file. Timeseries data is stored in an additional set of files you specify in timeseries_pointers.csv. Documentation for each of the files is found below:

Required Files

bus.csv

This file is used to define buses. Add one row for each bus in the system. Each row in the CSV file will cause a bus dictionary object to be added to `['elements']['bus']` in the Egret model.

Each row with a non-zero *MW Load* and/or non-zero *MVAR Load* will also cause a load to be added to `['elements']['load']` in Egret, and each row with a non-zero *MW Shunt G* and/or non-zero *MVAR Shunt B* will cause a shunt to be added to `['elements']['shunt']` in Egret.

Table 1: bus.csv Columns

Column Name	Description	Egret
Bus ID	A unique string identifier for the bus. This string is used to refer to this bus in other CSV files.	Not used by Egret except during parsing of CSV files.
Bus Name	A human-friendly unique string for this bus.	Used as the bus name in Egret. Data for this bus is stored in a bus dictionary stored at `['elements']['bus'][<Bus Name>]`. This is also the name of the load, if a load is added for the bus (a load is added if MW Load or MVAR Load is non-zero). The load dictionary is stored at `['elements']['load'][<Bus Name>]`. This is also the name of the shunt, if a shunt is added for the bus (a bus is added if *MW Shunt G* or *MVAR Shunt G* is non-zero). The shunt dictionary is stored at `['elements']['shunt'][<Bus Name>]`.
BaseKV	The bus base voltage. Must be non-zero positive.	Stored in the Egret bus dictionary as `base_kv`.
Bus Type	The type of bus. Can be one of the following: • *PQ* • *PV* • *Ref*	Stored in Egret bus dictionary as `matpower_bustype`. The *Ref* bus type is stored in Egret in all lower case (*ref*).
MW Load	Magnitude of the load on the bus.	Stored in the Egret bus dictionary as `p_load`. A non-zero value causes a load to be added; see *Bus Loads*.
MVAR Load	Magnitude of the reactive load on the bus.	Stored in the Egret bus dictionary as `q_load`. A non-zero value causes a load to be added; see *Bus Loads*.
V Mag	Voltage magnitude setpoint	Stored in the Egret bus dictionary as `vm`.
V Angle	Voltage angle setpoint in degrees	Stored in the Egret bus dictionary as `va`. If the Bus Type is *Ref*, this value must be *0.0*.
Area	The area the bus is in.	Stored in the Egret bus dictionary as `area`. An area dictionary is added to the Egret model for each unique area mentioned in the file. The Egret area dictionary is found at `['elements']['area'][<Area>]`. See *Areas*.
Zone	The zone the bus is in.	Stored in the Egret bus dictionary as `zone`.
MW Shunt G	Optional.	Stored in the shunt dictionary as `gs`. See *Shunts*.
MVAR Shunt B	Optional.	Stored in the shunt dictionary as `bs`. See *Shunts*.

Additional Bus Values

The following values are automatically added to the bus dictionary:

- `v_min` = *0.95*
- `v_max` = *1.05*

Bus Loads

If a bus has a non-zero *MW Load* or *MVAR Load*, a load dictionary is added to Egret at ['elements']['load'][<*Bus Name*>]. The load dictionary will have the following values taken from bus.csv:

- `bus` = *Bus Name*
- `p_load` = *MW Load*
- `q_load` = *MVAR Load*
- `area` = *Area*
- `zone` = *Zone*

An additional property is automatically added, always with the same value:

- `in_service` = *true*

Loads can (and usually do) vary throughout the study horizon. Variable loads are defined using a timeseries (see *timeseries_pointers.csv*).

Shunts

If a bus has a non-zero *MW Shunt G* or a non-zero *MVAR Shunt B*, a shunt dictionary is added to Egret at ['elements']['shunt'][<*Bus Name*>]. The shunt dictionary will have the following values taken from bus.csv:

- `bus` = *Bus Name*
- `gs` = *MW Shunt G*
- `bs` = *MVAR Shunt B*

An additional property is automatically added, always with the same value:

- `shunt_type` = *fixed*

Areas

Each unique area mentioned in bus.csv leads to an area being created in the Egret model at ['elements']['area'][<*Area*>], using the area name as it appears in bus.csv.

branch.csv

This file defines branches - flow pathways between pairs of buses - including lines and transformers. Add a row for each branch in the system. Each row in the CSV file will cause a branch dictionary to be added to `['elements']['branch']` in the Egret model.

Table 2: ListTable

Column Name	Description	Egret
UID	A unique string identifier for the branch.	Used as the branch name in Egret. Data for this branch is stored in a branch dictionary stored at `['elements']['branch'][<UID>]`.
From Bus	The *Bus ID* of one end of the branch	The *Bus Name* of the bus with the corresponding *Bus ID*, as entered in bus.csv, is stored in the Egret branch dictionary as `from_bus`.
To Bus	The *Bus ID* of the other end of the branch	The *Bus Name* of the bus with the corresponding *Bus ID*, as entered in bus.csv, is stored in the Egret branch dictionary as `to_bus`.
R	Branch resistance p.u.	Stored in Egret bus dictionary as `resistance`.
X	Branch reactance p.u.	Stored in the Egret bus dictionary as `reactance`.
B	Charging susceptance p.u.	Stored in the Egret bus dictionary as `charging_susceptance`.
Cont Rating	Continuous flow limit in MW	Stored in the Egret bus dictionary as `rating_long_term`. Optional.
LTE Rating	Non-continuous long term flow limit in MW	Stored in the Egret bus dictionary as `rating_short_term`. Optional.
STE Rating	Short term flow limit in MW	Stored in the Egret bus dictionary as `rating_emergency`. Optional.
Tr Ratio	Transformer winding ratio.	If non-zero, branch is treated as a transformer. If blank or zero, branch is considered a line. See *Lines and Transformers* below.

Additional Branch Values

The following values are automatically added to every branch dictionary in the Egret model:

- `in_service` = *true*
- `angle_diff_min` = *-90*
- `angle_diff_max` = *90*
- `pf` = *null*
- `qf` = *null*
- `pt` = *null*
- `qt` = *null*

Lines and Transformers

Each branch is either a line or a transformer. The type of branch is determined by the *Tr Ratio*. If this field is blank or zero, the branch is a line and the following property is added to the branch dictionary:

- `branch_type` = *line*

If the *Tr Ratio* is a non-zero value, the following properties are added to the branch dictionary:

- `branch_type` = *transformer*
- `transformer_tap_ratio` = *Tr Ratio*
- `transformer_phase_shift` = *0*

gen.csv

This file is where generators are defined. Add one row for each generator in the model, including both thermal and renewable generators.

Table 3: gen.csv Columns

Column Name	Description	Egret
GEN UID	A unique string identifier for the generator.	Used as the branch name in Egret. Data for this branch is stored in a generator dictionary stored at `['elements']['generator'][<GEN UID>]`.
Bus ID	*Bus ID* of connecting bus	The *Bus Name* of the bus with the matching *Bus ID*, as entered in bus.csv, is stored in the Egret generator dictionary as `bus`.
Unit Type	The kind of generator	Typically stored in `unit_type`. Has additional side effects. See *Generator Types* below.
Fuel	The type of fuel used by the generator	Stored in the generator dictionary as `fuel`
MW Inj	Real power injection setpoint	Stored in the generator dictionary as `pg`
MVAR Inj	Reactive power injection setpoint	Stored in the generator dictionary as `qg`
PMin MW	Minimum stable real power injection	May be left blank. If present, stored in the generator dictionary in multiple places: `p_min`, `startup_capacity`, `shutdown_capacity`, and `p_min_agc`
PMax MW	Maximum stable real power injection	May be left blank. If present, stored in the generator dictionary in multiple places: `p_max` and `p_max_agc`
QMin MVAR	Minimum stable reactive power injection	May be left blank. If present, stored in the generator dictionary as `q_min`
QMax MVAR	Maximum stable reactive power injection	May be left blank. If present, stored in the generator dictionary as `q_max`
Ramp Rate MW/Min	Maximum ramp up and ramp down rate	Thermal generators only. May be left blank. If present, stored in the generator dictionary in multiple places: `ramp_q` and `ramp_agc`
Output_pct_0 through *Output_pct_<N>*	The fraction of *PMax MW* for fuel curve point *i* (See *Fuel Curves* below).	Thermal generators only. See *Fuel Curves* below.
HR_avg_0	Average heat rate between 0 and the first fuel curve point, in BTU/kWh	Thermal generators only. See *Fuel Curves* below.
HR_incr_1 through *HR_incr_<N>*	Additional heat rate between fuel curve point *i-1* and fuel curve point *i*, in BTU/kWh.	Thermal generators only. See *Fuel Curves* below.
Fuel Price $/MMBTU	Fuel price in Dollars per million BTU	Thermal generators only. Stored in the generator dictionary as `fuel_cost`.
Non Fuel Start Cost $	Dollars expended each time the generator starts up.	Thermal generators only. Stored in the generator dictionary as `non_fuel_startup_cost`.
Min Down Time Hr	Minimum off time required before unit restart	Thermal generators only. Stored in the generator dictionary as `min_down_time`.
Min Up time Hr	Minimum off time required before unit restart	Thermal generators only. Stored in the generator dictionary as `min_up_time`.
Start Time Cold Hr	Time since shutdown after which a cold start is required	Thermal generators only. See *Startup Curves* below
Start Time Warm Hr	Time since shutdown after which a warm start is required	Thermal generators only. See *Startup Curves* below
Start Time Hot Hr	Time since shutdown after which a hot start is required	Thermal generators only. See *Startup Curves* below
Start Heat Cold MBTU	Fuel required to startup from cold	Thermal generators only. See *Startup Curves* below
Start Heat Warm MBTU	Fuel required to startup from warm	Thermal generators only. See *Startup Curves* below
Start Heat Hot MBTU	Fuel required to startup from hot	Thermal generators only. See *Startup Curves* below

Additional Generator Values

The following values are automatically added to all generator dictionaries:

- `in_service` = *true*
- `mbase` = *100.0*
- `area` = Area of the bus identified by *Bus ID*
- `zone` = Zone of the bus identified by *Bus ID*

If the generator is a thermal generator, these additional values are also added:

- `agc_capable` = *true*
- `shutdown_cost` = *0.0*
- `ramp_up_60min` = *60* * `ramp_q`
- `ramp_down_60min` = *60* * `ramp_q`

Generator Types

The *Unit Type* column determines whether the generator will be treated as thermal or renewable, or if the generator will be skipped.

If the *Unit Type* is *Storage* or *CSP*, the generator is skipped and left out of the Egret model.

If the Unit Type is *WIND*, *HYDRO*, *RTPV*, or *PV*, then these values are set:

- `generator_type` = *renewable*
- `unit_type` = *Unit Type*

If the *Unit Type* is *ROR*, then these values are set:

- `generator_type` = *renewable*
- `unit_type` = *HYDRO*

For all other values of *Unit Type*, these properties are set:

- `generator_type` = *thermal*
- `unit_type` = *Unit Type*

Fuel Curves

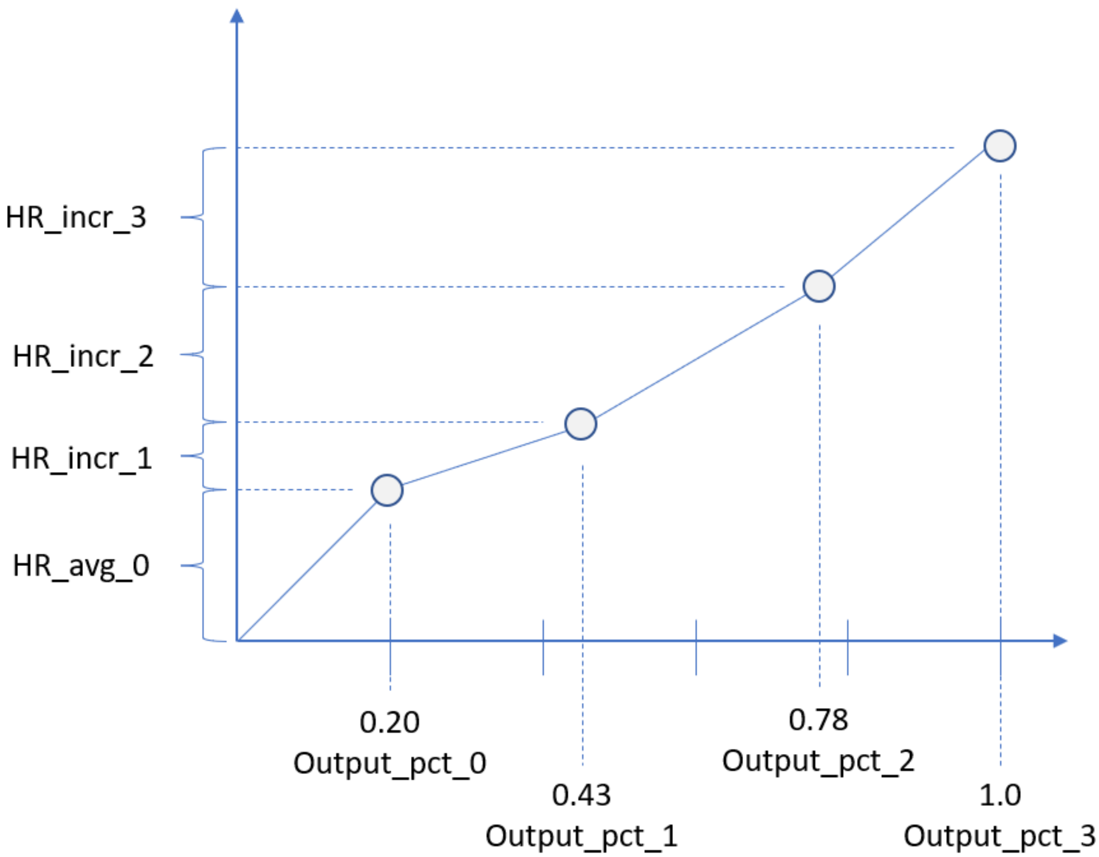
Fuel curves describe the amount of fuel consumed by the generator when producing different levels of power. A fuel curve is defined by a set of points, where each point identifies a power output rate and the amount of fuel required to generate that amount of power.

Power output rates are defined by the *Output_pct_<N>* columns, such as *Output_pct_0*, *Output_pct_1*, and so on. You can include any number of *Output_pct_<N>* columns, but they must be numbered sequentially (0, 1, 2, and so on, up to the desired number of fuel curve points). The value of each *Output_pct_<N>* column is a fraction of the maximum real power output (*PMax MW*), ranging from 0 to 1. Values must be in ascending order: *Output_pct_1* must be greater than *Output_pct_0*, *Output_pct_2* must be greater than *Output_pct_1*, and so on.

Corresponding fuel requirements are defined by the *HR_avg_0* column (for fuel curve point 0) and by *HR_incr_<N>* columns (for fuel curve points 1 and above). *HR_avg_0* is the fuel required to achieve *Output_pct_0*. *HR_incr_1* is the amount of *additional* fuel (the fuel increment) required to achieve *Output_pct_1*, *HR_incr_2* is the amount of additional fuel required to go from *Output_pct_1* to *Ouput_pct_2*, and so on. The fuel consumption curve is required to be convex above point 0; the slope of lines between fuel curve points must increase as you move to the right. Values of *HR_incr_** must be chosen to reflect this requirement.

Within each row, the number of non-blank *HR_** columns must must match the number of non-blank *Output_pct_<N>* columns. However, different rows can have different numbers of points in their fuel curves. Columns beyond the number of points in the fuel curve should be left blank.

The diagram below shows an example of a fuel curve with 4 points. The output percentage increases along the X-axis with each successive point. Fuel consumption values on the Y-axis are calculated by adding fuel increments to the previous Y values. Note that the fuel consumption curve is convex above *Output_pct_0*.



Fuel curves are stored in the Egret generator dictionary as `p_fuel`. Values in the fuel curve are in MW (rather than output percent) and MMBTU/hr (rather than BTU/kWh). Fuel costs are calculated by interpolating the fuel curve for the current output rate, then multiplying by the `fuel_cost`.

Startup Curves

Startup curves define the amount of fuel required to start a generator, based on how long it has been since the generator was shut off.

- If the time since the generator was most recently shut down is less than either the *Min Down Time Hr* or the *Start Time Hot Hr*, the generator cannot yet be restarted.
- If the time since shutdown is at least *Min Down Time Hr* and *Start Time Hot Hr*, but less than *Start Time Warm Hr*, then the generator can do a hot start, consuming *Start Heat Hot MMBTU*.
- If the time since shutdown is at least *Start Time Warm Hr*, but less than *Start Time Cold Hr*, then the generator can do a warm start, consuming *Start Heat Warm MMBTU*.
- If the time since shutodown is at least *Start Time Cold Hr*, then the generator can do a cold start, consuming *Start Heat Cold MMBTU*.

reserves.csv

This file defines the reserve products to be included in the model. Reserve products impose requirements on surplus generation capacity within a particular area under certain conditions. Each reserve product has a category and an area. The reserve product's category identifies the conditions under which its requirements apply, and its area identifies the region where the requirements apply.

There are 5 supported reserve product categories. The table below shows the name of reserve product categories on the left as they appear in CSV input files, and the corresponding name in Egret on the right.

Table 4: Reserve Product Categories

CSV Reserve Product Category	Egret reserve product name
Spin_Up	spinning_reserve_requirement
Reg_Up	regulation_up_requirement
Reg_Down	regulation_down_requirement
Flex_Up	flexible_ramp_up_requirement
Flex_Down	flexible_ramp_down_requirement

Each reserve product's category and applicable area are embedded in its name, as *<category>_R<area>*. For example, a spinning reserve requirement for an area named *"Area 1"* would be named *"Spin_Up_RArea 1"*.

Table 5: reserves.csv Columns

Column Name	Description	Egret
Reserve Product	The name of the reserve product, following the *<category>_R<area>* naming convention.	Added to the area's Egret dictionary as the Egret reserve product name.
Requirement (MW)	Magnitude of the reserve requirement. This value is ignored if there is a timeseries associated with the reserve product.	If honored, it is used as the value of the Egret reserve product name entry in the area's dictionary.

Reserve Requirement Magnitudes

The magnitude of each reserve requirement may be constant throughout the entire simulation, or it may change as specified by a timeseries in timeseries_pointers.csv. If the magnitude is constant, enter it in this file as the *Requirement (MW)*. If it varies during the study period, associate a timeseries with the reserve product (see *timeseries_pointers.csv*). In this case, the magnitude entered in this file is discarded and is replaced with appropriate timeseries values.

Applicability to RUCs and SCEDs

Each category of reserve product may be configured to apply to RUC plans, to SCED operations, or both. This is designated in simulation_objects.csv. See that file's documentation for details.

simulation_objects.csv

This file is used to enter data about the data set as a whole. Each row specifies a global parameter with two values, one that applies to forecasts and another that applies to real-time data (actuals). The file has three columns:

Table 6: simulation_objects.csv Columns

Column Name	Description
Simulation_Parameters	Which global parameter is set by this row
DAY_AHEAD	The row's value for forecast data and/or RUC plans
REAL_TIME	The row's value as it applies to real-time data and/or SCED operations

The following values of *Simulation_Parameter* are supported:

Table 7: Supported values of *Simulation_Parameter* in simulation_objects.csv

Simulation_Parameter	Required?	Parameter Description	DAY_AHEAD	REAL_TIME
Period_Resolution	Yes	The number of seconds between values in timeseries data files	The number of seconds between values in *DAY_AHEAD* timeseries data files	The number of seconds between values in *REAL_TIME* timeseries data files
Date_From	Yes	The date and time of the first value in each timeseries data file. Most reasonable formats are accepted.	The date and time of the first value in each DAY_AHEAD timeseries data file	The date and time of the first value in each REAL_TIME timeseries data file
Date_To	Yes	The latest date and time for which we have enough data in timeseries files to formulate a RUC or SCED. Most reasonable formats are accepted. See *Date_To Details* below.	The latest date and time for which we have enough data in DAY_AHEAD timeseries data files to formulate a RUC	The latest date and time for which we have enough data in REAL_TIME timeseries data files to formulate a SCED
Look_Ahead_Periods_Per_Step	Yes	The default number of look-ahead periods to use in RUC or SCED formulations	The default number of look-ahead periods to use in RUC formulations	The default number of look-ahead periods to use in SCED formulations
Look_Ahead_Resolution	Yes	The default number of seconds between each look-ahead period used in RUC or SCED formulations. See *Look_Ahead_Resolution Details* below.	The default number of seconds between each look-ahead period in RUC formulations	The default number of seconds between each look-ahead period in SCED formulations formulations
Reserve_Products	No	Which reserve products to enforce for RUC plans or SCED operations. See *Reserve_Products Details* below.	Which reserve products to enforce for RUC plans	Which reserve products to enforce for SCED operations

Date_To Details

The value of *Date_To* identifies the latest time for which there is enough data to formulate or RUC (for *DAY_AHEAD*) or SCED (for *REAL_TIME*), including look-ahead periods. This is not the date and time of the final value in timeseries data files. Instead, the *Date_To* is *Look_Ahead_Periods_Per_Step* * *Look_Ahead_Resolution* before the date and time of the final value in the timeseries data files.

For example, consider a data set with 24 look-ahead periods with a look-ahead resolution of 1 hour. If the final value in a timeseries is for April 10^{th} at midnight, then *Date_To* is April 9^{th} at midnight, because that is the latest time for which we have enough data to satisfy the 24 hour look-ahead requirement.

Look_Ahead_Resolution Details

The *Look_Ahead_Resolution* parameter is used to determine the date and time of the final value in timeseries data files, as described in *Date_To Details*. Despite its name, it is not used to specify the look-ahead resolution used during simulation. The actual look-ahead resolution used during simulation is determined by configuration parameters passed to Prescient. Prescient will interpolate the available data as necessary to honor the look-ahead resolution specified in its configuration parameters.

Reserve_Products Details

Some categories of reserve products may apply to RUC formulations, while others may apply to SCED formulations. This row allows you to configure which reserve product categories apply to each formulation type. Reserve product categories listed in the *DAY_AHEAD* column impose their requirements on RUC formulations, and reserve product categories listed in the *REAL_TIME* column impose their requirements on SCED formulations.

Specify applicable reserve product categories as a comma-separated list. Only listed reserve product categories will be imposed on corresponding formulations. Supported reserve products are *Spin_Up*, *Reg_Up*, *Reg_Down*, *Flex_Up*, and *Flex_Down*.

This row is optional. If you leave the row out, all reserve categories apply to both RUCs and SCEDs.

Optional Files

timeseries_pointers.csv

This file identifies where to find timeseries values, and which model elements they apply to. Each row in the file identifies a model element (such as a particular generator's power output, or an area's load), whether the values are forecast or actual values, and what file holds the values. The CSV file has the following columns:

Table 8: timeseries_pointers.csv Columns

Column Name	Description
Simulation	Either *DAY_AHEAD* or *REAL_TIME*. If *DAY_AHEAD*, the values are forecasts that inform RUC formulations. If *REAL_TIME*, the values are actual values used in SCED formulations.
Category	What kind of object the data is for. Supported values are: • *Generator* • *Area* • *Reserve*
Object	The name of the specific object the data is for
Parameter	The specific attribute of the object that the data is for
Data File	The path to the file holding the timeseries values.

The model element the data applies to is identified by the *Category*, *Object*, and *Parameter*. Which parameters are supported depend on the *Category*.

- If *Category* is *Generator*, then *Object* must be the name of a generator as specified in the *GEN UID* column of gen.csv. *Parameter* must be either *PMax MW* or *PMin MW*.
- If *Category* is *Area*, then *Object* must be an area name referenced in bus.csv, and *Parameter* must be *MW Load*. The timeseries values specify the load imposed on the area at each timestep.

- If *Category* is *Reserve*, then *Object* is a reserve product name in *<category>_R<area>* format, and *Parameter* must be *Requirement*. The timeseries values specify the magnitude of the reserve requirement for the reserve product.

The *Data File* is the path to the CSV file holding timeseries values. The path can be relative or absolute. If it is relative, it is relative to the folder containing timeseries_pointers.csv.

Timeseries File Formats

There are two supported formats for timeseries files, columnar and 2D. A columnar file has a row for each value in the timeseries, while a 2D file has a row for each day and a column for each value within the day. A columnar file can have multiple data columns for each row, allowing data for multiple model elements to be stored in the same file. A 2D file can only hold a single timeseries.

Both file formats store data at equally spaced time intervals. Each day is split into periods, numbered 1 through N. The first period of each day starts at midnight. The duration of each period is specified by the Period_Resolution row in simulation_objects.csv. The number of periods per day must add up to 24 hours per day. Note that *DAY_AHEAD* periods and *REAL_TIME* periods often have different durations, so the appropriate the number of periods per day may depend on whether the data are forecasts or actuals.

Each file's data must cover the time period from *DATE_FROM* to *DATE_TO*, as specified in simulation_objects.csv, including the extra look-ahead periods after *DATE_TO*.

Columnar Timeseries Files

A columnar timeseries file has one row per period. It has 4 columns that identify the date and period of the row's data, followed by any number of data columns. The name of each data column must match the name of the object the data pertains to, such as the name of the appropriate generator. Here is an example of the first few rows of a columnar timeseries file with data for two generators named *Hydro1* and *Hydro2*:

Table 9: Example Columnar Timeseries File

Year	Month	Day	Period	Hydro1	Hydro2
2023	4	1	1	2.0152	11.958
2023	4	1	2	2.3055	12.616
…	…	…	…	…	…

Note that the *Year*, *Month*, *Day*, and *Period* are entered as integer values.

2D Timeseries Files

A 2D timeseries file holds data for a single timeseries in a 2D layout. The file has *Year*, *Month*, and *Day* columns, followed by one column per period in each day. For example, a file with hourly data will have 27 columns: the *Year*, *Month*, and *Day* columns followed by 24 period columns:

Table 10: Example 2D Timeseries Data File

Year	Month	Day	1	2	…	24
2023	4	1	1.989	2.0152	…	1.958
2023	4	1	2 .015	2.3055	…	2.616
…	…	…	…	…	…	…

The name of each period column must be the period number, from 1 to N.

dc_branch.csv

This file is where DC branches are defined. Prescient has limited support for DC branches, as indicated by the small number of columns in this file.

This file is optional; if the file does not exist, no DC branches are added to the model. If the file exists, add a row for each DC branch in the model. Each row in the file will cause a DC branch dictionary to be added to `['elements']['dc_branch']` in the Egret model.

Table 11: dc_branch.csv Columns

Column Name	Description	Egret
UID	A unique string identifier for the DC branch.	Used as the branch name in Egret. Data for this branch is stored in a branch dictionary located at `['elements']['dc_branch'][<UID>]`.
From Bus	The *Bus ID* of one end of the branch	The *Bus Name* of the bus with the matching *Bus ID*, as entered in bus.csv, is stored in the Egret branch dictionary as `from_bus`.
To Bus	The *Bus ID* of the other end of the branch	The *Bus Name* of the bus with the matching *Bus ID*, as entered in bus.csv, is stored in the Egret branch dictionary as `to_bus`.
MW Load	Power Demand in MW	This value is repeated 3 times in the Egret dc_branch dictionary, as `rating_short_term`, `rating_long_term`, and `rating_emergency`.

initial_status.csv

This file holds the initial state of each generator. It is an optional file; defaults are used if the file is not present. The file contains a header row and 1 to 3 data rows.

The header row consists of one column per generator, with the column name being the name of the generator, as specified in the *GEN UID* column of gen.csv.

The first data row is the status of each generator at the start of the simulation period, where a positive number indicates how many time periods the generator has been running, and a negative number indicates how many time periods since the generator was shut down. The first row must contain a value for every generator.

The second data row is the power output of each generator in the time period just before the start of the simulation. This row can be left blank for all generators, or should be populated for all generators.

The third data row is the reactive power of the generator in the time period just before the start of the simulation. This row can be left blank, or should be populated for all generators. If the second row was left blank, then the third row must also be left blank. In other words, the third row can hold data only if the second row also holds data.

4.1.2 Overview of Input Data

Data is read into Prescient from a collection of CSV files in a format similar to that used by RTS-GMLC. See *The CSV Input File Format* for details. The data in CSV input files includes the definition of the system under study including system elements such as generators and buses, as well as timeseries data such as variable loads and renewable generator outputs.

Timeseries data in CSV files often covers a larger time period than is included in the study. The dates provided to Prescient as configuration parameters are used to trim down the data read from input files.

Internally, Prescient stores data in the Egret format.

4.2 Python Classes and Functions

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search