TESP Documentation

Release 1.0

Pacific Northwest National Laboratory

Mar 25, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION TO TRANSACTIVE ENERGY AND TESP

1.1 What Is Transactive Energy?

Let's start from the beginning: what is transactive energy? Though there are many definitions the one we'll use here comes from the GridWise Architecture Council [24]

> A system of economic and control mechanisms that allows the dynamic balance of supply and demand across the entire electrical infrastructure using value as a key operational parameter.

Transactive energy seeks to allow all actors in the electrical energy system to participate in the dynamic management of the power system, most fundamentally, the moment-by-moment balancing of supply of electrical energy and its demand. Typically, the balancing is done by only the largest actors in the system: the large bulk power system generators and their customers, the utility companies (sometimes with the help of an intermediary market and system manager). These interactions take on various forms from long-term contracts to auctions every few minutes and effectively form the wholesale electricity market.

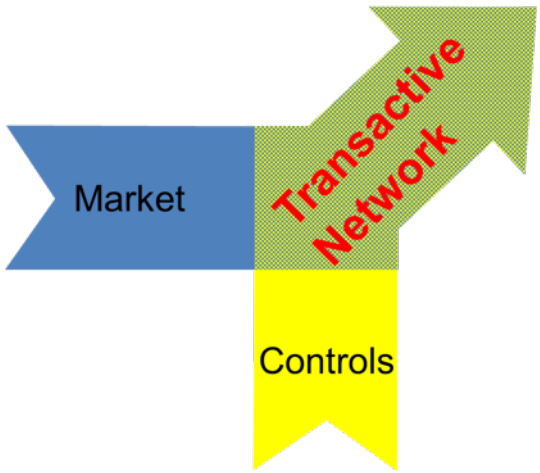
Most end users of that energy, though, are not part of this interaction and are generally ignorant of it. We may experience some efforts by our local utility to include us through time-of-use tariffs where the price of energy changes in predictable ways throughout the day or by peak-period pricing where the cost of energy rises dramatically for a few hours a few times a year. These kinds of mechanisms are muted attempts to incentivize energy customers of the state of the power system with higher retail prices intended to follow the general trend of wholesale prices. But because these retail tariffs are pre-defined once and used for years to come, they have no ability to accurately reflect the state of the power system on any given day.

Transactive energy seeks to change that by including all actors, including the end customers, in the management of the power system by providing appropriate value signals to all participants such that the dynamic needs of the power system can be addressed appropriately by all participants. If electrical energy is in short supply, a higher price to customers can incentivize them to reduce consumption until further resources can be provisioned or the peak in demand has passed. Conversely, when prices are low, consumers are signaled that energy-intensive activities (*e.g.* charging electric vehicles, running pool pumps) will be less costly and will not place an undue stress on the power system.

By choosing to enable all participants in the management through appropriate value signals, transactive energy hopes to use the flexibility in all existing actors behavior to manage the power system in a more efficient manner

1.2 Transactive Energy Simulation Platform (TESP)

Transactive Systems Program



Traditionally, analysis of power systems has been split cleanly between the bulk power system (*e.g.* long-distance transmission lines connecting large generation to large load centers like cities) and the distribution system (*e.g.* neighborhoods). This separation has been motivated by different needs. The bulk power system is often concerned with finding the most economical means of dispatching generators to meet the expected load or trying to determine the most economical expansion of the power system. Distribution system planners have been concerned with appropriately sizing the power lines the run through a neighborhood or what measures need to be made to ensure good voltage management.

Transactive energy, by trying to include all actors as participants in the management of the system, necessarily breaks down these analysis barriers and ties these two distinct analysis domains togethers. For appropriate analysis of a transactive system, the analysis needs to allow participation of the management of the system by all actors which means the analysis tools need to be able to represent all actors in appropriate ways. Furthermore, the models of the participants need to be more fully fleshed out so that large loads (*e.g.* air conditioners, EV chargers, water heaters) can be modeled in way that allows their loadshape to be altered as actors respond to value signals. And depending on the particular analysis, further models that were not previously used may be needed such as those or rooftop solar panels or community energy storage.

Given these more complex analysis requirements, a more complex simulation technique was needed: co-simulation. Co-simulation allows the dynamic integration of multiple simulation tools such that the outputs of each can can be used as inputs to others. For example, the voltage at a particular transmission bus that is found when the power flow is solved for the bulk power system can be used as the substation voltage when the distribution system needs to solve its power flow. Conversely, the distribution system load can be fed back up to the bulk power system simulation for use when it solves it's power flow.

Co-simulation allows the analysis of more complex and larger scale power system problems than would be possible otherwise but comes with the cost of complexity. The individual simulation tools used in the co-simulation need to integrated into the co-simulation platform so they can send and receive messages with other tools. Each tool needs to be configured to not only use the appropriate models but also to send and receive the correct messages. The data coming out of all the simulation tools needs to be synthesized and analyzed to form conclusions.

The Transactive Energy Simulation Platform (TESP) has been developed by Pacific Northwest National Laboratory (PNNL) under funding and direction by the United States Department of Energy to minimize the barriers of transactive energy analysis (the complexities of co-simulation being chief among them) to allow for more efficient and effective analysis of potential transactive energy schemes. Specifically, TESP aims to provide:

- Appropriate simulation tools to model common transactive scenarios
- Integrated simulation tools into a co-simulation platform
- Generic models and other input datasets that may be needed for transactive analysis.
- Demonstrations of the various co-simulation capabilities
- Fully realized examples of transactive analysis
- All of the above in an easily managed software package that is readily customizable and altered for particular analysis needs.

1.3 TESP Software Stack Overview

Fig. 1.1 shows the typical co-simulation software stack when using TESP for TE analysis.

- GridLAB-D covers the electric power distribution system [8] and residential buildings (OpenDSS is a similar alternative).
- PYPOWER, MATPOWER/MOST or AMES covers the bulk power system and the transmission system operator (TSO) [1, 28].
- EnergyPlus covers large commercial buildings [12]
- ns-3 is a communication system simulator that can also host software agents.
- The integrating message bus, using either the Hierarchical Engine for Large-scale Infrastructure Co-Simulation (HELICS [20]) manages the time step synchronization and message exchange among all of the federated simulation modules.

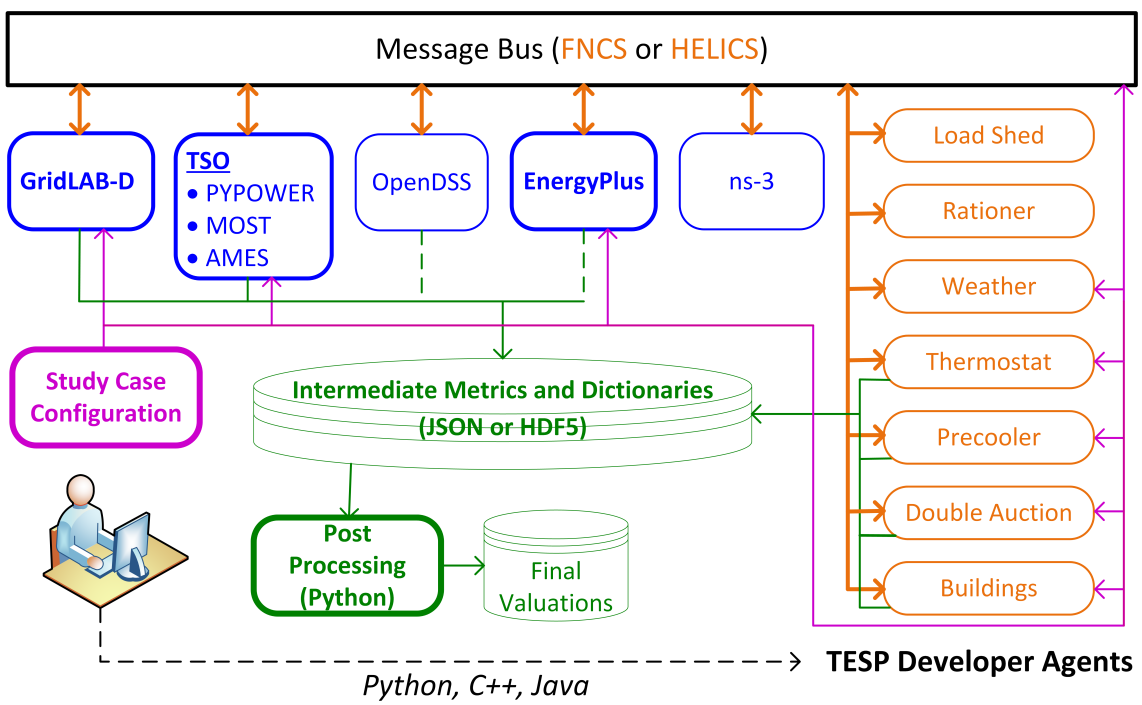


Fig. 1.1: TESP Rev 1 components federated through FNCS and/or HELICS.

Assuming this software stack satisfies the needs of the particular analysis, the user interacts with TESP by configuring simulation cases (magenta) and then running them. Simulation federates or Agents, write intermediate outputs and metadata (green), which the user plots, post-processes and analyzes to reach conclusions.

(Some of of the simulators and agents in Fig. 1.1 have to be configured by hand. OpenDSS writes output in its native, non-TESP format, and EnergyPlus writes output only through the Buildings agent; these are indicated with dashed green lines. The ns-3 simulator doesn't write output; it's presently used in just one example, for which the GridLAB-D outputs are adequate.)

Most of the Agents in Fig. 1.1 were implemented in the Python programming language, though custom code for TESP can also be implemented in other languages like C++ and Java. To demonstrate, the Buildings agent was implemented in C++ and one version of one of the examples distributed with TESP (Load Shed) has an agent was written in Java.

1.4 Overview of Transactive Energy Analysis Process

Given the complexity of many TE analysis and the variety of software components that may need to be used to perform said analysis, taking time to clearly plan the analysis conceptually and practically will generally save time in the long run. The following is an outline of the process PNNL has developed and implemented for TE Analysis.

1.4.1 Value Model

As TE is fundamentally built on the concept of value transactions or exchanges, developing a value model that explicitly shows this can be helpful. These models are able to clearly show which system actors will be modeled in the TE analysis, which ones are outside the system but involved in the value exchanges and which values are being exchanged through the operation of the TE system.

With the value exchanges modeled, it is much easier to identify and define relevant performance mechanisms for the TE system. Is an actor giving up comfort to save money (for example by adjusting a thermostat during a high-price period)? If so, defining a metric to measure how much discomfort the actor is enduring could be important. How far from the desired setpoint does the thermostat go? Are there times when a maximum or minimum setpoint is reached? And how much money does the actor save by responding to this dynamic price? These metrics will be the measure by which the TE system is evaluated and should be clearly related to the value model. Furthermore, generally, they should be able to be calculated in both the transactive case and the base or business-as-usual case. If this is not the case, it is likely a sign that the metrics have not been entirely thought through.

Finally, prior to writing any code, it is worth developing a flowchart or sequence diagram of how the TE system (or even all simulated activities) will operate. This flowchart helps provide clarity of how and when the value exchanges will take place and the process by which each actor accrues value. It will also serve as a good starting place when writing the code to realize the TE system.

An example of these models can be found in the *Valuation Model*.

1.4.2 Design of Analysis

With a value model in place and the fundamental of the TE system outlined, the question then becomes one of methods and means: what needs to be done to achieve the analysis goals? For TE studies, co-simulation will likely be a part of the answer but is likely to be far from complete. It would not be unusual for new input datasets to be needed by various entities in the co-simulation. There may be specific values that need to be defined either for the co-simulation (*e.g.* renewable penetration level) or for use in post-processing the data (*e.g.* assumed cost of solar panels in the year of the analysis).

Regardless, the critical element are the performance metrics that have been previously defined. These metrics define specific input data and the goal of the analysis is to produce those values. Some of these may come directly from the co-simulation but it would not be unusual for many of them to defined by separate analysis or from relevant literature.

These data are used by a series of analysis steps, one after the other, to produce the required inputs for the final metrics. Develop a plan for this analysis workflow is helpful in not only ensuring that all the data that is needed has been accounted for but also helping to guide scoping decisions and being clear about where the extra effort may be needed to achieve the analysis goals.

To show the impact of the TE system, to demonstrate the impacts of the system the design should make it clear in some way what defines the base or business-as-usual case and what constitutes the transactive case(s). Keeping the system models and inputs constant across the cases makes a direct apples-to-apples comparison possible in the key performance metrics.

Lastly, in addition to the key performance metrics, there are likely to be supplemental data that is helpful in validating the performance of the co-simulation and the analysis as a whole. These validation metrics would not generally be defined by the value model because they generally are not tied to the value flows. For example, if the TE system adjusts air-conditioning thermostats higher during high price periods and lower as the price drops a validation graph could be created to show the thermostat setpoint throughout the day with the energy price overlaid. Though this graph and its associated data are not necessarily needed to calculate the final value-based metrics it is useful to confirm that the co-simulation that produced this data is working as expected.

An example of these models can be found in the *Analysis Design Model*.

1.4.3 Co-Simulation Implementation and Execution

With an analysis plan in place, now the direct work of implementation can begin. The analysis plan should clearly show the analysis steps that are required (*e.g.* writing new transactive agent code, finding input data sets, writing scripts for calculating final metrics).

The co-simulation will be run at some point and this may require computation resources beyond what a typical desktop or laptop computer provides. There may need to be some extra work done in developing deployment plans and tools for the co-simulation components. Relatedly, the datasets produced by the co-simulation could be very large and requires more complex data handling and storage techniques.

1.4.4 Post-Processing and Analysis

With the final dataset produced from all the necessary analysis steps the validation and key performance metrics can be calculated and reviewed. Ideally the presentations of the data show both that the co-simulation and the analysis as a whole have been constructed correctly (validation) and that the TE system is having the expected impact. Both the validation and the value-based metrics should have comparisons between base and transactive case(s) making the impact of the transactive system clear.

1.5 Next Steps After TESP-Based Analysis

TESP is well-suited when trying to explore, design, and validate the fundamental principles of a new or modified transactive mechanism. Having simulation results that show the mechanism works as expected and/or produces specific value is an important first step but is by no means the last. Generally, simulations in TESP are not concerned with appropriate access to data, data imperfections, protocols for the messages, and many other implementation details.

Users who wish to validate or demonstrate transactive mechanisms under operational constraints may consider GridAPPS-D. GridAPPS-D provides a standards-based API for grid operations and control applications (or apps) to interface with a simulation-based field emulator. Apps interact with the emulated as they would with an operating system in the field; i.e., asynchronously without direct access to co-simulation variables. GridAPPS-D provides app development tools, a library of services, and a standard model set to facilitate app development. Model management and configuration capabilities and a test manager enable app validation under emulated abnormal grid conditions. For more information visit the GridAPPS-D homepage and/or repository.

CHAPTER

TWO

INSTALLING AND BUILDING TESP

TESP, as a software platform, provides much of its functionality through third-party software that it integrates to provide the means of performing transactive analysis. All of this software is open-source and in every case can be built on any of the three major OSs (Mac, Windows, and Linux). That said, TESP itself is only officially supported on Ubuntu Linux simply as a means of reducing the support burden and allowing us, the TESP developers, to add and improve TESP itself without spending the significant time required to ensure functionality across all three OSs. If you're comfortable with building your own software, a quick inspection of the build scripts we use to install TESP on Ubuntu Linux will be likely all you need to figure out how to get it built and installed on your OS of choice.

The current supported method uses a set of custom build scripts to download source code from public repositories and build from source. This particular method was chosen for a key reason: it allows you, the user, to pull down the latest version of TESP (which may include bug fixes in a special branch) and have those changes quickly be realized in your installation. Similarly, the live linking of the third-party tools' repositories with git allows similar bugfix changes and upgrades to those tools to be readily applied to your installation. This installation method provides not only working executables of all the software but also all of the source code for said tools. In fact, for those that are more daring or have more complex analysis requirements, this installation method allows edits to the source code of any of the software tools and by re-compiling and installing that source (which the installation scripts automate) a custom version of any of the tools can be utilized and maintained in this installation. (We'll cover this in more detail in a dedicated section on customizing TESP in *Developing and Customizing TESP*.)

2.1 Create a Github account (somewhat optional)

Many of the repositories holding the source code for the simulation tools used in TESP are hosted on Github. If you want to be able to push code back up to these repositories, you'll need a Github account. The Github user name and email are typically provided as part of running the TESP install script but are technically optional and can be omitted. TESP will still install but the ability to commit back into the repository will not exist.

2.2 Installation Guide

This guide will assume that TESP is being installed on a clean Ubuntu Linux installation or Windows 10 using WSL2.

For many, this will be a virtual machine (VM) and the good news is that there is a no-cost means of creating this VM using Oracle's VirtualBox. Other commercial virtualization software such as VMWare and Parallels will also do the trick.

For Windows 10 users we can use WSL2. In many ways a it is like a virtual machine that allows shell commands just as if it were Linux.

2.2.1 Creating a Ubuntu Linux VM with VirtualBox

There is lots of documentation out there on installing Ubuntu on a VirtualBox VM and we won't rehash those instructions here. Below are a few links you can try:

- Install Ubuntu on Oracle VirtualBox
- How to Install Ubuntu on VirtualBox? Here's the Full Guide
- How to install Ubuntu on VirtualBox

You can get the OS disk image (.iso) from Ubuntu and mount it in the virtual machine for installation. Alternatively, OSboxes provides a hard drive image with the OS already installed that you can install in your virtual machine.

A few notes:

- Installing TESP will require building (compiling) software from source which is generally resource intensive. Giving the VM lots of compute resources (CPUs, memory) will be very helpful when installing (and running) TESP.
- However you install Ubuntu, there is a good chance that some of the software included in the installation is out of date since the image was made. Ubuntu should prompt you to update the software but if it doesn't manually run the "Update Software" application, otherwise TESP install will do for you.
- Make sure you install the VirtualBox Guest Additions to improve the integration with the host OS and the overall user experience.
- Administrative access for the account where TESP will be installed is required.

2.2.2 Creating a WLS2 on Windows 10

The setup procedure for creating a WLS2 on Windows 10 very easy with these instructions . However, some further adjustments maybe necessary with permissions and proxy.

2.2.3 Running TESP install script

Once you have a working Ubuntu/WLS2 on Windows installation, the TESP install process is straight-forward. From a command prompt, issue the following commands:

Listing 2.1: TESP installation commands for Ubuntu/WLS2 on Window10

```
wget --no-check-certificate https://raw.githubusercontent.com/pnnl/tesp/main/scripts/
↪tesp.sh
chmod 755 tesp.sh
./tesp.sh <Github user name> <Github email address>
```

In the last line, the optional name and email must be entered in that order, both must be included. For me, it looks like this:

Listing 2.2: TESP sample installation script execution

```
./tesp.sh trevorhardy trevor.hardy@pnnl.gov
```

Running this script will kick off a process where all latest linux packages are installed, then the Python environment is setup with the required packages installed after that the repositories are cloned locally and compiled one-by-one. Depending on the computing resources available and network bandwidth, this process will generally take a few hours.

Due to this length of time, *sudo* credentials will likely expire at one or more points in the build process and will need to be re-entered.

After getting TESP software installed and the executables built, the TESP installation will have created a *tesp* directory the same directory level as the *tesp.sh* script. All installed files are descended from the *tesp* directory.

2.2.4 Setting Up TESP Environment

Prior to running any of the included examples, we need to be sure to set up the compute environment so that the TESP software can be found by the system. The *tespEnv* file is added at the same level as the root *tesp* folder and it contains all the environment configuration.

```
source tespEnv
```

You will need to do this every time you open a new terminal. If the computing environment set-up you're using allows it, you can add this command to your ".bashrc" or equivalent so that it is automatically run for you each time you start a terminal session.

2.2.5 Validate TESP installation

Once the installation process has finished there should be a folder name *tesp* where all the TESP software, data, and models have been installed. There are several progressively more comprehensive ways to validate the TESP installation process.

Check OS can find TESP software

TESP includes a small script that attempts to run a trivial command with each of the software packages it installs (typically checking the version). The script is located at *~/grid/tesp/scripts/build/versions.sh*. This script runs automatically at the end of the build and install process and produces and output something like this (version numbers will vary):

```
++++++++++++++  Compiling and Installing TESP software is complete!  ++++++++++++++

TESP software modules installed are:

TESP 1.3.0
FNCS installed
HELICS 3.4.0-main-g0b3d894e7 (2023-10-03)
HELICS Java, 3.4.0-main-g0b3d894e7 (2023-10-03)

GridLAB-D 5.1.0-19475 (4ea6109e:develop:Modified) 64-bit LINUX RELEASE
EnergyPlus, Version 9.3.0-fd4546e21b (No OpenGL)
NS-3 installed
Ipopt 3.13.2 (x86_64-pc-linux-gnu), ASL(20190605)

++++++++++++++  TESP has been installed! That's all folks!  ++++++++++++++
```

If you see any messages indicating *command not found* if indicates one of the software packages did not install correctly.

Check directory structure

An easy manual high-level check to see if TESP installed correctly is to look at the directory structure that was installed and make sure everything ended up in the right place. A tree view from top-level *tesp* folder you should see something like this:

```
grid
├── tesp
├── venv
│   ├── bin
│   │   ├── ...
│   │   ├── python
│   │   ├── python3
│   │   └── python3.8
│   ├── man
│   ├── include
│   ├── lib
│   │   └── python3.8
│   └── share
│       ├── man
│       └── doc
├── tenv
│   ├── bin
│   │   ├── eplus_agent*
│   │   ├── fncs*
│   │   ├── gridlabd*
│   │   ├── helics*
│   │   ├── ipopt*
│   │   ├── mini_federate
│   │   ├── ns3-*
│   │   └── test_comm
│   ├── energyplus
│   │   ├── ...
│   ├── include
│   │   ├── coin-or
│   │   ├── fncs.h
│   │   ├── fncs.hpp
│   │   ├── gridlabd
│   │   ├── helics
│   │   └── ns3-dev
│   ├── java
│   │   ├── fncs.jar
│   │   ├── helics-2.8.0.jar
│   │   ├── helics.jar -> helics-2.8.0.jar
│   │   ├── libhelicsJava.so
│   │   └── libJNIfncs.so
│   ├── lib
│   │   ├── cmake
│   │   ├── gridlabd
│   │   ├── libcoinasl.*
│   │   ├── libcoinmumps.*
│   │   ├── libfncs.*
│   │   ├── libhelics*
```

(continues on next page)

(continued from previous page)

```
│   │   ├── libipoptamplinterface.*
│   │   ├── libipopt.*
│   │   ├── libns3*
│   │   ├── libsipopt.*
│   │   └── pkgconfig
│   └── share
│       ├── doc
│       ├── gridlabd
│       ├── helics
│       ├── java
│       └── man
└── repo
    ├── Ames-V5.0
    ├── EnergyPlus
    ├── fncs
    ├── gridlab-d
    ├── HELICS-src
    ├── Ipopt
    ├── KLU_DLL
    ├── ns-3-dev
    ├── ThirdParty-ASL
    └── ThirdParty-Mumps
```

Shorter Autotest: Example Subset

A (relatively) shorter autotest script has been written that tests many (but not all) of the installed examples to verify the installation was successful. This test can be run as follows and assumes the commandline prompt '~$' in the TESP root directory:

Listing 2.3: TESP example subset autotest

```
~$ source tespEnv
(TESP) ~$ cd ~grid/tesp/examples
(TESP) ~/grid/tesp/examples$ exec python3 autotest.py &> short.log &
(TESP) ~/grid/tesp/examples$ deactivate
~/grid/tesp/examples$
```

The first command is essential after starting a terminal session prior to running anything in TESP for the first time. After running the first line above, the prompt now shows the prefix *(TESP)* being used for the variable environment. If you don't run the first line, simulations will generally fail for lack of being able to find their dependencies. If things aren't working, double-check to make sure your commandline shows the prefix *(TESP)*.

The forth command, 'deactivate', returns the environment path to the state it was before the the first command started and remove the *(TESP)* prefix from the prompt. All other environment variables are present but the TESP python requirements may/may not be present, depending on your configuration.

The commandline that ran this autotest was executed in the background, so that can close the terminal, but don't close the VM. You can open terminal later and check progress by viewing the short.log. Even this subset of examples can take several hours to run (roughly 4.9 hours in the results shown below) and at the end, prints a final results table showing the runtime in seconds for each test:

```
Test Case(s)                        Time Taken
==========================================
GridLAB-D Player/Recorder             0.891868
Loadshed - HELICS ns-3                4.129848
Loadshed - HELICS Python              1.014755
Loadshed - HELICS Java                4.055216
Loadshed - HELICS/EPlus              11.930494
Establishing baseline results        70.629668
Load shedding w/o comm network       70.957783
Load shedding over comm network     368.501483
PYPOWER - HELICS                      5.283039
Houston,TX Baselines build types   1183.537504
Generated EMS/IDF files - HELICS      1.555593
EnergyPlus EMS - HELICS               6.210505
Weather Agent - HELICS                8.205831
Houses                              180.550353
TE30 - HELICS Market                297.990520
TE30 - HELICS No Market             301.580143
4 Feeders - HELICS                  824.673296
Eplus w/Comm - HELICS               432.880265
No Comm Base - HELICS              3289.462584
Eplus Restaurant - HELICS          2958.467020
SGIP1c - HELICS                    3087.110814
```

Total runtime will depend on the compute resources available and each example run serially.

Longer Autotest: Remaining examples

Listing 2.4: TESP remaining examples autotest

```
~$ source tespEnv
(TESP) ~$ cd ~/grid/tesp/examples
(TESP) ~/grid/tesp/examples$ exec python3 autotest_long.py &> long.log &
```

The commandline that ran this autotest was executed in the background, so that can close the terminal, but don't close the VM. You can open terminal later and check progress by viewing the long.log. This subset of examples can take several days to run (roughly 49.8 hours in the results shown below) and at the end, prints a final results table showing the runtime in seconds for each test:

```
Test Case(s)                        Time Taken
==========================================
SGIP1a - HELICS                    14132.360023
SGIP1b - HELICS                    14143.111387
SGIP1c - HELICS                    15305.805641
SGIP1d - HELICS                    17289.504798
SGIP1e - HELICS                    19784.953376
SGIP1ex - HELICS                   19623.103407
PNNL Team IEEE8500                     0.023067
PNNL Team 30 - HELICS                 98.790637
PNNL Team ti30 - HELICS              103.635829
PNNL Team 8500 - HELICS            13872.056659
PNNL Team 8500 TOU - HELICS        13375.151752
```

(continues on next page)

(continued from previous page)

```
PNNL Team 8500 Volt - HELICS   13513.567733
PNNL Team 8500 Base            12338.000525
PNNL Team 8500 VoltVar         13278.476238
PNNL Team 8500 VoltWatt        12584.246679
```

2.2.6 Trouble-shooting Installation (forthcoming)

CHAPTER

THREE

TESP DEMONSTRATIONS AND EXAMPLES

To help users of TESP to better understand how the software platform has been created and integrated, a number of sample projects are included in the distribution of TESP and are divided into two categories: capability demonstrations and analysis examples.

Capability demonstrations are sample projects that are relatively simple and intended to show off a single or very small number of features of TESP. They may be a demonstration of the use of one of the third-party tools and its integration into TESP or one of the custom agents that are provided with TESP. These demonstrations are not legitimate analysis in and of themselves and the results from them are not intended to provide any significant insight into good transactive system design principles or behaviors.

In contract, analysis examples are versions of analysis that have been performed in the past with TESP with specific analysis objectives. These examples have much more comprehensive documentation within TESP and have produced one or more publications that provide further detail. The versions of these analysis that are included in TESP are not necessarily the same as those that were originally used but they are very similar and are examples of specific transactive concepts or mechanisms. The results of the version of these examples that are distributed with TESP are not only examples of how a transactive energy study could be assembled with TESP but the results produced by running the examples will be as meaningful (though not necessarily identical) to those used to produce the original analysis conclusions and publications.

3.1 TESP Capability Demonstrations

3.1.1 loadshed

Co-Simulation Architecture

This directory contains Python and Java versions of a loadshed example on the 13-bus IEEE test feeder, modeled in GridLAB-D. In this model, a stand-alone external controller (*helicshed.py* and *helicshed.java*) send "OPEN" and "CLOSED" commands to a switch in the GridLAB-D model (*loadshed.glm*) through a simple two-node communication model in ns-3 (*loadshedCommNetwork.cc*).

Running the Demonstration

loadshed - verify GridLAB-D, ns-3 and Python over HELICS

```
cd ~/grid/tesp/examples/capabilities/loadshedh
./clean.sh # Removes any left-over results and log files
./runhpy.sh
./plot.sh
```

loadshed - Python without ns-3

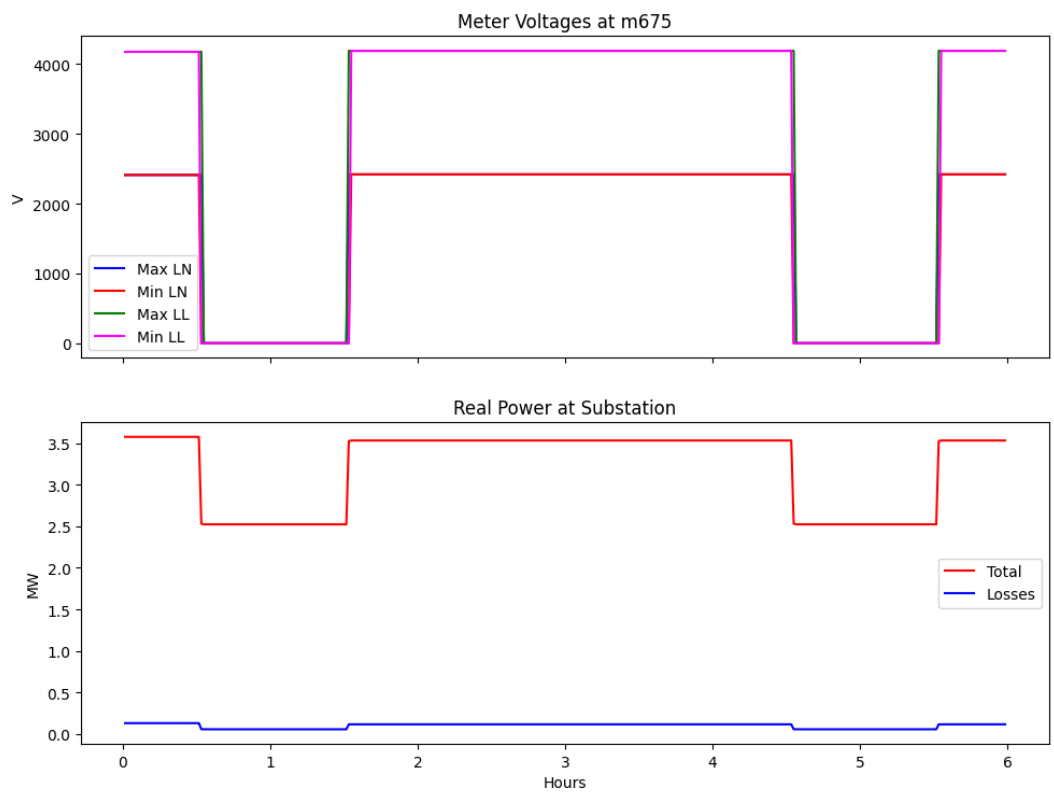
```
cd ~/grid/tesp/examples/capabilities/loadshedh
./clean.sh # Removes any left-over results and log files
./runhpy0.sh
./plot.sh
```

loadshed - verify GridLAB-D, ns3 and Java over HELICS

```
cd ~/grid/tesp/examples/capabilities/loadshedh
./clean.sh
./runhjava.sh
./plot.sh
```

Results

Running any of the above versions and plotting the results will yield the following graph.



File Listing

It differs from the other examples, in not using the *tesp_support* Python package. Instead, three local source files have been provided as possible starting points in developing your own source files in Python or Java.

- *clean.sh* - shell script that deletes any existing results and log file in the current directory.
- *helics_gld_msg0.json* - GridLAB-D configuration file when running without ns-3.
- *helics_gld_msg.json* - GridLAB-D configuration file when running with ns-3.
- *helics_gld_msg_no_pub.json*

- *helics_gld_msg_old_island.json*
- *helicshed0.py* is the same loadshedding agent, implemented in Python for HELICS. Test with *runhpy0.sh*
- *helicshed.java* is the same loadshedding agent, implemented in Java for HELICS. Test with *runhjava.sh*
- *helicshed.py* is the same loadshedding agent, implemented in Python for HELICS with ns-3. Test with *runhpy.sh*
- *helicsRecorder.json* - HELICS configuration file for the helics_recorder used to capture the switch commands.
- *loadshedCommNetwork.cc* - ns-3 federate source code. Note that ns-3 logging is enabled only if ns-3 was built in debug mode.
- *loadshedCommNetworkConfig.json* - HELICS configuration file for the ns-3 federate.
- *loadshedConfig.json* - HELICS configuration file for the Python or Java federate
- *loadshed_dict.json*
- *loadshed.glm* - GridLAB-D model of the IEEE 13-bus feeder containing the switch being controlled by the Python or Java controllers.
- *Makefile* - defines the build process for the ns-3 model
- *plot_loadshed.py* - plotting program for the simulation results
- *plot.sh* - shell script used to plot the results
- *README.rst* - This file
- *runjava.sh* - launcher script for running the loadshed demo using a Java loadshed agent.
- *runhpy0.sh* - launcher script for running the loadshed demo using a Python agent without using the ns-3 communication network model.
- *runhpy.sh* - launcher script for running the loadshed demo using a Python agent include the ns-3 communication model.

3.1.2 loadshed - Prototypical Feeder with Point-to-Point Communication Network

This particular version of the *loadshed* example offers the ability to build a point-to-point communication network for a prototypical feeder [21]. It has been developed to introduce a manageable communication network modeled in ns-3, which could include as nodes any of the nodes in a populated prototypical feeder. This allowed to build a communication network of smaller or larger size to study the scalability of the ns-3 models for large power distribution systems applications, and the impact the size of the communication network has on their performance.

Scope of the example

This particular example wants to demonstrate the following capabilities of TESP:

- Integrate a communication simulator, that is ns-3, that would allow modelling the cyber communication layer of a distribution system.
- Allow for a customizable communication network model built through an ns-3 model considering the distribution topology as an entry point.
- Demonstrate how the communication network structure affects the expected response of the distribution system, due to latencies and distance between communication nodes.

Co-Simulation Architecture

The directory structure for this example follows the structure:

- *R1-12.47-1* folder contains:
  - *R1-12.47-1_processed.glm* - the populated prototypical feeder GridLAB-D model, obtained by running the *feederGenerator.py* script;
  - *R1-12.47-1_gridlabd.json* - the HELICS configuration file for the GridLAB-D model containing the subscription, publications, or the end points that allow the GridLAB-D federate to interact with other federates. In this particular case, several end points corresponding to particular loads in the system publish their current power demand and subscribe to the command on their connection status, that is either to stay *IN_SERVICE* or go *OUT_OF_SERVICE* when load needs to be dropped.
  - *recorders.glm* - the recorder objects to save measurements in corresponding CSV files, at the level of each of the nodes communicating through HELICS.
  - Folders to store the results of the simulations in CSV format, that is the content of the files written using the recorder objects in GridLAB-D. These folders have been manually created to save and distinguish among the different simulated scenarios:
    - *outputs_noNS3_noLoadShed* contains the results of a stand alone GridLAB-D simulation, in which the system is not subject to any external commands to shed load.
    - *outputs_noNS3_LoadShed* contains the results of the scenario in which ideal communication is established between the Python federate emulating load shed control and GridLAB-D, that is there will be no delay between the moment the command to shed load is initiated and the corresponding load is actually disconnected from the grid and/or later reconnected.
    - *outputs_withNS3_LoadShed* contains the results of the simulation run using a cyber-communication network following the distribution network topology to route the load shedding commands from the substation level down to the loads.
- *R1-12.47-1-substation* folder contains:
  - *R1-12.47-1_substation.py* - the substation federate running at the level of the distribution model substation node, monitoring points in the network, and deciding when and what loads should be dropped.
  - *R1-12.47-1_HELICS_substConf.json* - the HELICS configuration file for the substation federate.
  - *loadshedScenario.json* - the load shed scenario given in dictionary format to suggest when and what loads are to be taken offline and/or brought back online. For example, the following dictionary entry

```
{
  [...],
  "180":
  {
    "R1_12_47_1_tn_459_mhse_4": "OUT_OF_SERVICE",
    "R1_12_47_1_tn_506_mhse_1": "IN_SERVICE"
  },
  [...]
}
```

is interpreted by the substation federate to command at second 180 in the co-simulation to *R1_12_47_1_tn_459_mhse_4* to go offline, while *R1_12_47_1_tn_506_mhse_1* is brought online.

Caveat: The names of the assets in *loadshedScenario.json* file need to be exactly the same as the names in the GridLAB-D model, and in order for the outcome to be as expected, these particular assets need to be among the ones listed as HELICS subscribers for GridLAB-D model.

- *R1-12.47-1-communication* folder contains:
  - *loadshed-p2p-network.cc* - the ns-3 model that builds a point-to-point (p2p) network between a series of nodes in the distribution system that require to communicate. The model is written to allow for an interactive way of selecting which nodes of the distribution system to be nodes in the communication network. However, keep in mind that due to the linked hierarchy in the distribution network, some nodes might be mandatory to make sure there is a path between two communicating ones.

Caveat: Before running the co-simulation including the communication network, or any time after a modification is made to the model file, the model needs to be compiled using the provided *Makefile*, by running:

```
make clean
make
```

The following configuration parameters are given to the model through a JSON file, that is *R1-12.47-1_simConfig.json*:

```
{
  "Simulation": {
    "Simulation_Duration": 300,
    "Verbose": "true",
    "Case_Name": "R1-12.47-1_HELICS",
    "ns3_Network_Config": "./R1-12.47-1_ns3.json",
    "ns3_EP_Config": "./R1-12.47-1_HELICS_ns3Conf.json",
    "Anim_File": "./R1-12.47-1_HELICS_anim_P2P.xml",
    "Routing_File": "./R1-12.47-1_HELICS_route_P2P.xml",
    "Err_Log_File": "./R1-12.47-1_HELICS_P2P.log",
    "Node_Loc_File": "./R1-12.47-1_HELICS_P2P_nodes_reduced.txt",
    "Links_Loc_File": "./R1-12.47-1_HELICS_P2P_links.txt",
    "Node_List_File": "./R1-12.47-1_HELICS_P2P.lst"
  }
}
```

It specifies:

- The duration of the simulation in seconds as *Simulation_Duration*.
- Whether the ns-3 simulator should detail debugging information through the flag *Verbose*.
- A name for the model as *Case_Name*.
- The network configuration file as *ns3_Network_Config*. This file in JSON format has been built based on the distribution network topology and specifies all nodes in the system and how they are linked, depending on the case design.
- The ns-3 federate HELICS configuration file as *ns3_EP_Config*. This file lists the points in the communication network model that are going to participate in information exchange through HELICS.
- The output animation file as *Anim_File*, if the model is set to save an output of the network dynamics.
- The routing table output file as *Routing_File*, if the model is set to save the network routing table.
- An error logging file as *Err_Log_File* to account for possible mistakes in building the communication network.
- A list of all selected nodes with their names and location as *Node_Loc_File*.
- A list of all the links between the selected nodes as *Links_Loc_File*.
- A list of all existing node categories in the current feeder as *Node_List_File*.

Running the demonstration

The three scenarios studied comparatively in this example require running different numbers of simulators federated or not using HELICS. The following paragraphs details on how to run each of them. Keep in mind that the TESP repository already contains previously obtained results for this case study, which are presented below while introducing the co-simulation workflow.

Establishing baseline results

The first scenario establishes a baseline for the subsequent studies. It involves only the GridLAB-D model and therefore it can be run independently, without the use of a HELICS-based co-simulation platform.

1. Inside TESP, navigate to the folder containing the GridLAB-D model, that is *R1-12.47-1* folder in this case, e.g.

```
${HOME}/tesp/examples/capabilities/loadshed-prototypical-communication/R1-12.47-1
```

2. Inside this folder, at the terminal, run:

```
gridlabd R1-12.47-1_processed.glm
```

3. The following set of files are going to be generated inside the current directory:

- *substation_load.csv* - total load of the system measured at the substation level, at 1-second resolution, in *W*. The baseline for the load at the substation level is shown in Fig. 3.1 in blue.
- *R1_12_47_1_tn_15_mhse_1_rec.csv, R1_12_47_1_tn_128_mhse_2_rec.csv, R1_12_47_1_tn_459_mhse_4_rec.csv, R1_12_47_1_tn_506_mhse_1_rec.csv, R1_12_47_1_tn_564_mhse_4_rec.csv* - recorded load for several system loads in *W*, and their status (*IN SERVICE* or *OUT OF SERVICE*, meaning the load is connected to, or disconnected from the grid, respectively) at 1-second resolution. For the baseline case, all loads are considered to be connected to the grid the entire simulation period.

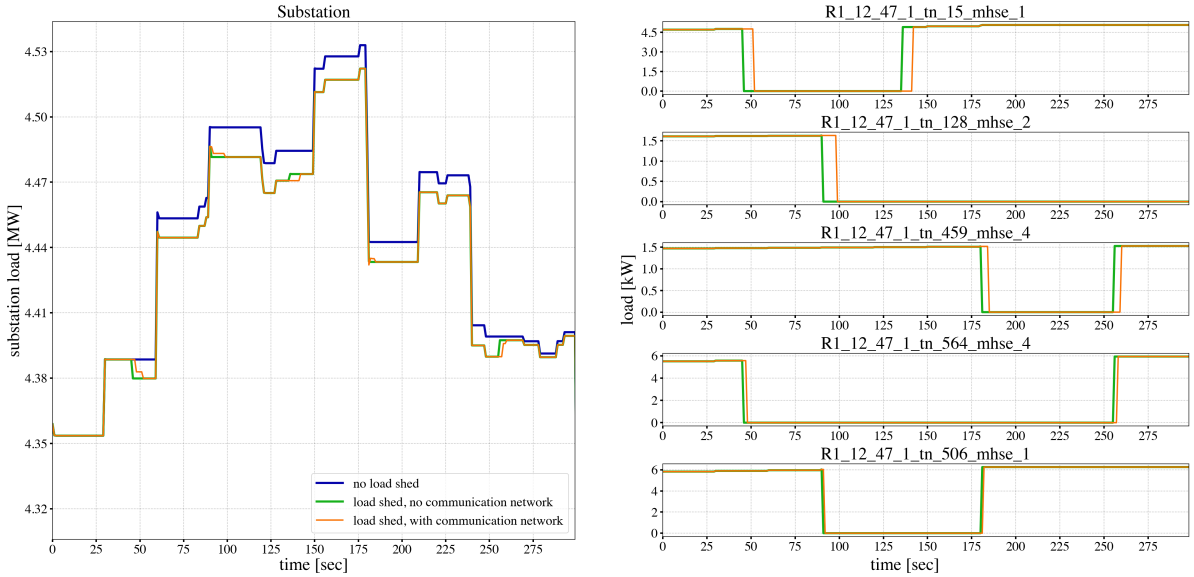


Fig. 3.1: Loadshed with prototypical feeder example results

Load shedding control without communication network

This scenario emulates a load shedding scenario where the decision to shed specific loads is taken at the substation level and the signals to disconnect and then, later, possibly re-connect loads are sent directly to the affected assets without engaging any communication infrastructure. This in the ideal case when no network latency is present. This scenario runs under TESP as a 2-federate co-simulation: the GridLAB-D running the feeder model, and a Python federate that sends the disconnect/connect signals to certain loads.

1. Inside TESP, navigate to the example folder, e.g.

```
${HOME}/tesp/examples/capabilities/loadshed-prototypical-communication
```

2. At the terminal, using the HELICS Command Line Interface (CLI), run

```
helics run --path=./R1-12.47-1_broker_conf_noNS3.json
```

The file *R1-12.47-1_broker_conf_noNS3.json* configures the co-simulation.

```
{
  "broker": true,
  "name": "LoadshedFederation",
  "federates": [
    {
      "name": "R1-12.47-1-federate",
      "host": "localhost",
      "directory": "./R1-12.47-1",
      "exec": "gridlabd -D USE_HELICS R1-12.47-1_processed.glm"
    },
    {
      "name": "R1-12.47-1-substation-federate",
      "host": "localhost",
      "directory": "./R1-12.47-1-substation",
      "exec": "python3 R1-12.47-1_substation.py --config R1-12.47-1_HELICS_substConf.
↪json --simTime 300"
    }
  ]
}
```

This configuration file identifies:

- The number of federates as the length of the *federates* vector (e.g. 2 in this case),
- Each federate with a name, specific folder to run in, and the command to execute to launch the federate.

Specifically, the Python federate emulating a control and decision center runs with:

- *–config* or *-c* flags followed by the HELICS configuration file *R1-12.47-1_HELICS_substConf.json*.
- *–simTime* or *-t* flags followed by an integer representing the number of seconds for how long the federate should run in co-simulation.

As seen in Fig. 3.1 in the right-hand side graphs in green, five loads are being disconnected from the grid at different times, and then some of them are reconnected. Because the control signal reaches the controlled loads instantaneously as there is no communication network between them and the substation, the distribution network sees a change in overall load immediately and as expected (depicted in green in the left-hand side graphs).

Load shedding control over communication network

The third scenario introduces ns-3 as the simulation federate for the communication layer realizing the cyber-connected node infrastructure of the distribution network. In this particular case, as shown in Fig. 3.2, there exists a realizable path from the substation to any other node in the distribution network and down to any load and DER (PV or battery).

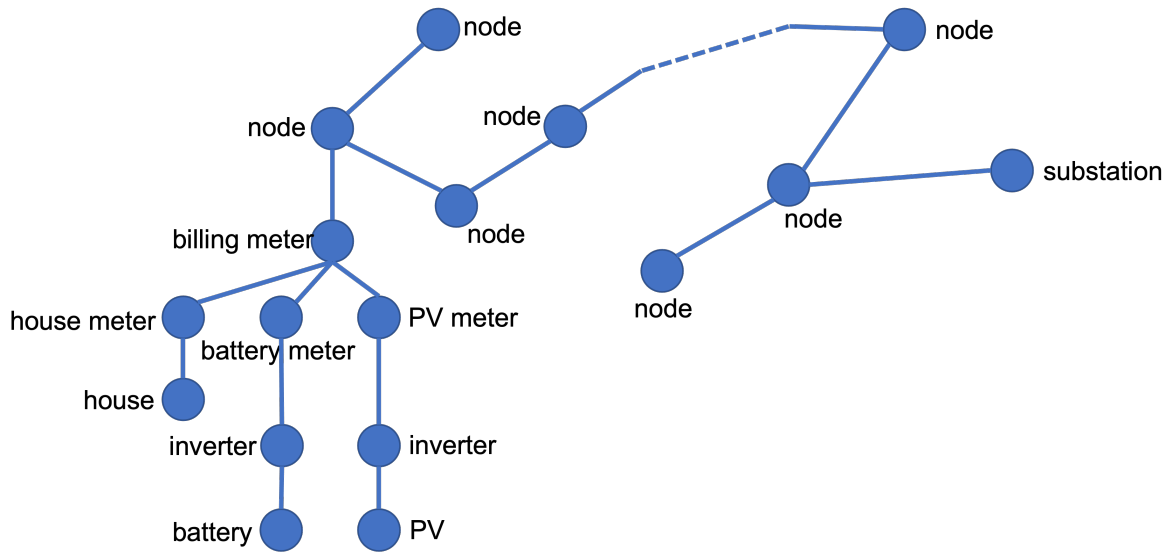


Fig. 3.2: Generic distribution network topology

The ns-3 model is set to either:

- Allow for interactive selection of which node category (substation, node, billing meter, house meter, house, battery meter, battery, solar meter, PV, or inverter) to be considered in building the ns-3 point-to-point network model, or
- Fix the communication network nodes through the ns-3 model file (requires re-compilation after any change made).

As this example requires control of the house loads from the substation level, the following node categories are fixed to be considered when building the ns-3 communication model to make sure each house load is reached from the substation:

- substation,
- node,
- billing meter,
- house meter.

This implies that from a total of 7,590 connected points in the distribution feeder (7,589 links), only 4,412 (4,411 links) are considered. The final communication network developed for this example considering the prototypical feeder *R1-12.47-1* in [21] is shown in Fig. 3.3, which also highlights the location of the substation and the 5 controlled loads.

To run this scenario, follow the steps:

1. Inside TESP, navigate to the example folder, e.g.

```
${HOME}/tesp/examples/capabilities/loadshed-prototypical-communication
```

2. At the terminal, using the HELICS Command Line Interface (CLI), run

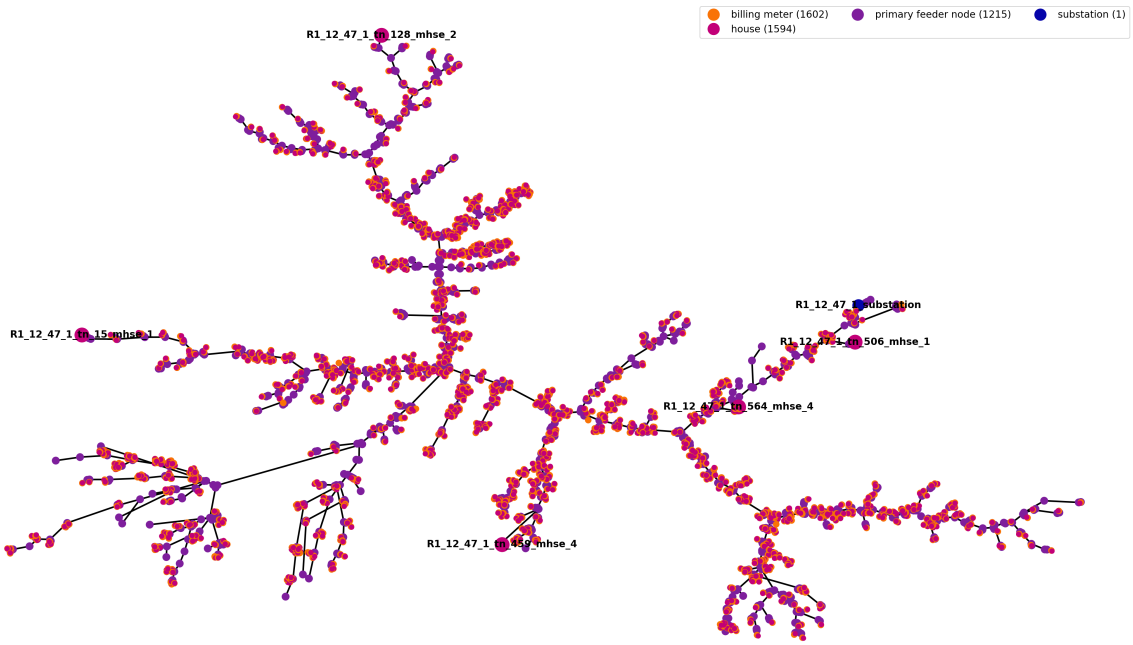


Fig. 3.3: Communication network topology

```
helics run --path=./R1-12.47-1_broker_conf_withNS3.json
```

The file *R1-12.47-1_broker_conf_withNS3.json* configures the co-simulation.

```
{
  "broker": true,
  "name": "LoadshedFederation",
  "federates": [
    {
      "name": "R1-12.47-1-federate",
      "host": "localhost",
      "directory": "./R1-12.47-1",
      "exec": "gridlabd -D USE_HELICS R1-12.47-1_processed.glm"
    },
    {
      "name": "R1-12.47-1-substation-federate",
      "host": "localhost",
      "directory": "./R1-12.47-1-substation",
      "exec": "python3 R1-12.47-1_substation.py --config R1-12.47-1_HELICS_substConf.
↪json --simTime 300"
    },
    {
      "name": "R1-12.47-1-communication",
      "host": "localhost",
      "directory": "./R1-12.47-1-communication",
      "exec": "./loadshed-p2p-network --simConfigFile=R1-12.47-1_simConfig.json"
    }
  ]
```

(continues on next page)

(continued from previous page)

```
}
```

The extra entry in this configuration file compared to the one in the previous scenario is related to the ns-3 federate that is set to run in the communication network folder with *R1-12.47-1_simConfig.json* as the federate configuration file.

In Fig. 3.1 the results of this scenario are shown in the orange color. In this example, for study purpose only, the delay on all point-to-point channels has been set to *100 ms*. This leads to a delayed response from the controlled loads going offline or online, as seen when compared to the their response when the control signals are not transmitted through a communication network. Moreover, when compared among the controlled loads, the latencies are variable as they also depend on the distance between the source and destination in the communication network and the number of hops the signal has to go through, fact corroborated by the physical distances between substation and controlled loads in Fig. 3.3.

3.1.3 PYPOWER Example

This example simply verifies that PYPOWER will run a 9-bus case and communicate over FNCS. To run and plot it:

```
./runpp.sh
python3 plots.py
```

In addition, traced FNCS messages will be written to pptracer.out

This example simply verifies that PYPOWER will run a 9-bus case and communicate over HELICS. To run and plot it:

```
./runhpp.sh
python3 plots.py
```

TODO: Player/Recorder comments

Directory contents:

- *clean.sh*; script that removes output and temporary files
- *helics_loads.txt*; HELICS player file for distribution loads system
- *helicsRecorder.txt*; HELICS recorder file for different system outputs
- *NonGLDLoad.txt*; text file of non-responsive loads on transmission buses
- *plots.py*; makes 1 page of plots for a case; eg 'python plots.py'
- *ppcase.json*; PYPOWER system definition
- *pptracer.yaml*; FNCS configuration for the message tracing utility
- *pypower.yaml*; FNCS configuration for PYPOWER
- *pypowerConfig.json*; HELICS configuration for PYPOWER
- *README.md*; this file
- *runhpp.sh*; script for running the case - HELICS
- *runpp.sh*; script for running the case - FNCS

3.1.4 weatherAgent

This weather agent needs an environment variable WEATHER_CONFIG to be set and point to the WeatherConfig.json file.

It reads in the csv weather data and the example data file is provided in this folder. If it is hourly data, the agent can do quadratic interpolation.

In order to match results from a TMY3 file containing the same measurements, it's necessary to provide GridLAB-D with the weather location's latitude, longitude and time zone meridian (in degrees, and < 0 if in the Western hemisphere). Initial values matching the start-time csv data may also be provided to GridLAB-D. See the *WeatherTester.glm* file in this directory for an example.

To run the example, invoke *./runh.sh* from a command prompt. This creates a weather data file from TMY3. Then it runs a GridLAB-D simulation two ways, once using the original TMY3 and again using the weather agent reading a data file. When the script completes, invoke *python3 compare_csv.py* from a command prompt; this verifies that both methods give essentially the same result.

The WeatherConfig.json sets the following parameters:

- *"name":* string, has to be "weather",
- *"StartTime":* string, start of the simulation, for example "2000-01-01 00:00:00",
- *"time_stop":* string, length of the simulation, unit can be "d, day, days, h, hour, hours, m, min, minute, minutes, s, sec, second, seconds", for example "70m",
- *"time_delta":* string, this is the time step that fncs broker registers and uses to find peer time for other federates, peer time is registered at handshake and cannot be changed by fncs::update_time_delta() function call, unit can be "d, day, days, h, hour, hours, m, min, minute, minutes, s, sec, second, seconds", for example "1s",
- *"publishInterval":* string, how often the agent publishes weather data, for example "5m",
- *"Forecast":* integer, 1: true for forecast; 0, NO forecast,
- *"ForecastLength":* string, how far into the future the forecast should be, for example "24h",
- *"PublishTimeAhead":* string, how much time ahead of the supposed publish time to publish the data, unit can be "d, day, days, h, hour, hours, m, min, minute, minutes, s, sec, second, seconds", for example "8s",
- *"AddErrorToForecast":* integer, 1: true; 0: false,
- *"broker":* string, has to be "tcp://localhost:5570",
- *"forecastPeriod":* integer, this is the period/cycle used in weather forecast to calculate the error, for example 48,
- ***"parameters":* are parameters needed in the weather forecast to add error, each weather factor could have different parameters, for now, only the parameters for temperature are set, the other ones do not have good tested parameters**
  - ***"temperature":* name of the factor, we also have humidity, solar_direct, solar_diffuse, pressure, wind_speed**
    - *"distribution":* 0: Uniform distribution; 1: Triangular distribution; 2: Truncated normal distribution
    - *"P_e_bias":* pu maximum bias at first hour, for example: 0.5,
    - *"P_e_envelope":* pu maximum error from mean values, for example: 0.08,
    - *"Lower_e_bound":* pu of the maximum error at the first hour, for example: 0.5

The following topics are published by the agent:

- weather/temperature

- weather/humidity
- weather/solar_direct
- weather/solar_diffuse
- weather/pressure
- weather/wind_speed
- weather/temperature/forecast
- weather/humidity/forecast
- weather/solar_direct/forecast
- weather/solar_diffuse/forecast
- weather/pressure/forecast
- weather/wind_speed/forecast

3.1.5 EnergyPlus Example

This example simply verifies that EnergyPlus will run a building model, and communicate over HELICS with an agent and message tracer. To run and plot it:

```
./runh.sh
python3 plots.py
```

In addition, traced messages will be written to recorder and log files.

Subdirectories:

- *eplusHelicsExample*; custom-built HELICS example for testing
- *forSchoolBase*; custom-built school dual controller EMS files for FNCS and HELICS
- *Windows*; helper scripts to run on Windows (no longer supported outside of Docker or VM)

File Directory:

- *archivedEms.idf*; original version of the custom-built school dual controller EMS (deprecated)
- *batch_ems_case.sh*; top-level script that simulates all reference buildings with EMS; calls run_seasonal_cases.sh
- *batch_plots.sh*; top-level script that plots all reference buildings with EMS; calls seasonal_plots.sh
- *bridge_eplus_agent.json*; configuration file for runfh_bridge.sh example (deprecated)
- *clean.sh*; script that removes output and temporary files
- *compile_png.py*; compiles all plots from batch_plots.sh into a Word document
- *eplus.yaml*; FNCS configuration for EnergyPlus
- *eplus_agent.yaml*; FNCS configuration for EnergyPlus agent
- *eplus_agentH.yaml*; HELICS configuration for EnergyPlus agent
- *eplusH.json*; HELICS configuration for EnergyPlus
- *helicsRecorder.json*; JSON configuration of the HELICS recorder
- *helicsRecorder.txt*; text configuration file for the HELICS recorder (deprecated)

- *kill23404.sh*; helper script that stops processes listening on port 23404 for HELICS (Linux/Mac)
- *kill5570.sh*; helper script that stops processes listening on port 5570 for FNCS (Linux/Mac)
- *make_all_ems.sh*; after run_baselines.sh, this script produces the EMS programs in separate IDF files
- *make_ems.sh*; makes a single EMS program from the contents of 'output', which typically comes from SchoolBase.idf
- *plots.py*; makes 1 page of plots for a case; eg 'python plots.py'
- *prices.txt*; sample price changes, published over FNCS to the building's transactive agent
- *README.md*; this file
- *run.sh*; Linux/Mac script for the case using FNCS (deprecated)
- *run2.sh*; FNCS version of the secondary school building the auto-generated EMS
- *run_baselines.sh*; simulate the reference buildings for one year, in preparation to make EMS programs
- *run_ems_case.sh*; runs a single HELICS-based reference building with given dates and price response
- *run_seasonal_cases.sh*; calls run_ems_case.sh for one reference building, summer and winter, with and without price response
- *runfh_bridge.sh*; runs FNCS EnergyPlus, bridged through a dual agent to HELICS federates (deprecated)
- *runh.sh*; Linux/Mac script for the case using HELICS
- *SchoolBase.idf* ; custom-built school building without the EMS
- *seasonal_plots.sh*;
- *tabulate_responses.py*;
- *tracer.yaml*; FNCS configuration for the message tracing utility

3.1.6 TE30 Demonstration

Co-Simulation Architecture

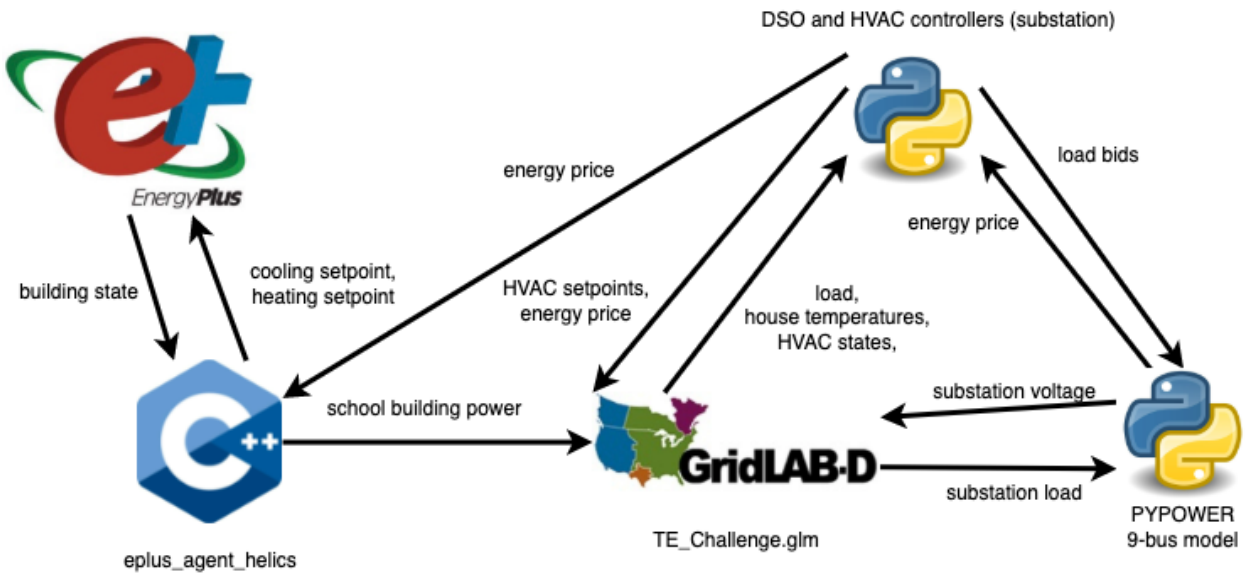
The TE30 demonstration was developed as part of NIST's TE Challenge This example file comprises 30 houses and a school building on a small, stiff, distribution circuit. It provides a medium-level test case for multiple HVAC transactive agents, with or without the double-auction real-time energy market.

The co-simulation data-exchange architecture is shown in the figure below. EnergyPlus is used the model a school that is attached to the distribution system and has a transactive agent that receives building state information and based on market conditions, adjusts the HVAC system's setpoints. The power consumption of the building is also sent to GridLAB-D.

GridLAB-D simulates the physics of the power system and and the houses, providing electrical state information to the bulk power system model and thermodynamic and electrical information on the houses in the model. Each house also has rooftop solar PV whose production is calculated by GridLAB-D. The distribution system operator (DSO) is modeled as an external python agent, and for performance reasons, the residential HVAC transactive agents are included in this code; this demonstration calls this the "substation" object.

The HVAC transactive controllers collect information on the state of their constituent houses and use that to form bids for the real-time energy market. The DSO (substation) aggregates these bids and presents them to the bulk power system real-time energy market which is run by PYPOWER. PYPOWER collects these bids and performs an economic

dispatch operation which defines the locational marginal price (LMP) for real-time energy at each transmission node, including the one to which the GridLAB-D model is attached; the LMP is communicated to the DSO. PYPOWER also calculates the physics of the bulk power system by running a powerflow which defines the voltage at the substation of the GridLAB-D model.



Running the Demonstration

NOTE: This example can take several minutes to run. After launching the appropriate "run" shell script check the status of the co-simulation by looking at the output of the "TE_ChallengeH0.csv" or "TE_ChallengeH.csv", depending on which run script is being used. This can be most easily done by running *tail -f TE_ChallengeH0.csv* and looking for new values to stop being written to the file.

TE30 without the transactive market

```
python3 prepare_case.py # creates additional configuration files, only needs to be run␣
↪once
./runh0.sh
python3 plots.py TE_ChallengeH0
```

TE30 with the transactive market

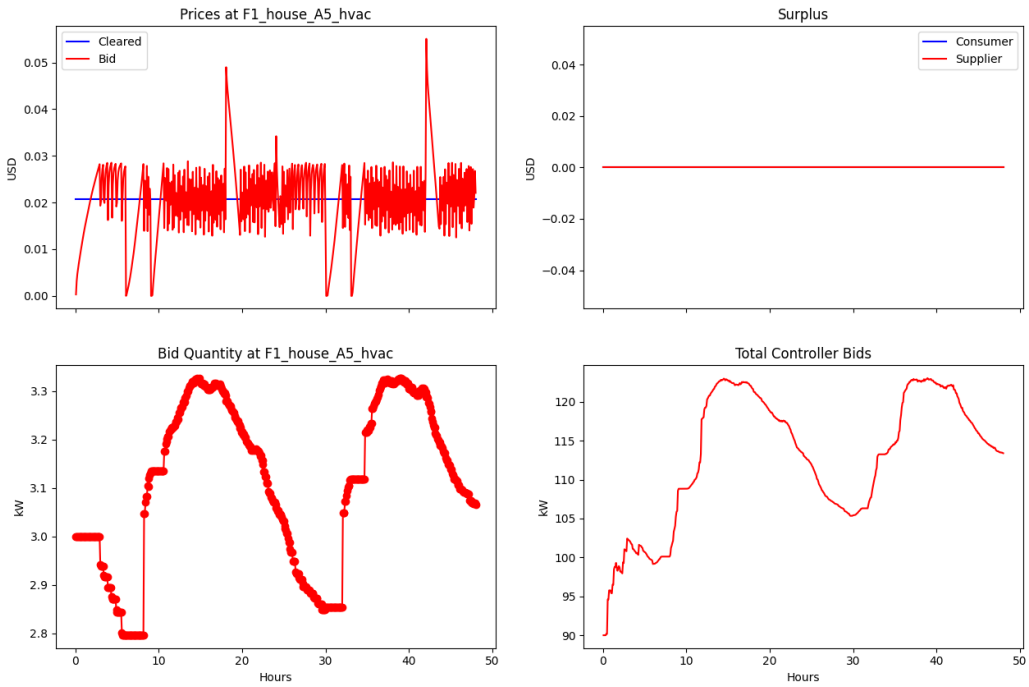
```
python3 prepare_case.py # not needed if already run for TE_ChallengeH0
./runh.sh
python3 plots.py TE_ChallengeH
```

Results

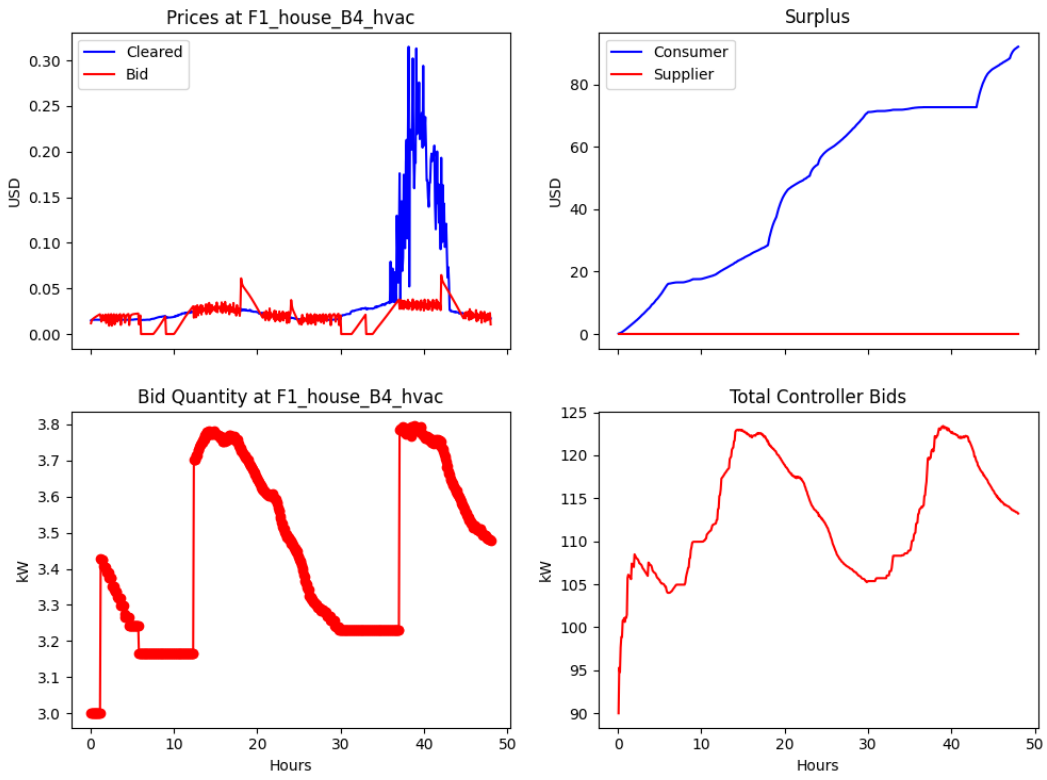
Running any of the above versions and plotting the results will yield the following graph.

Transactive HVAC Controllers

Base case

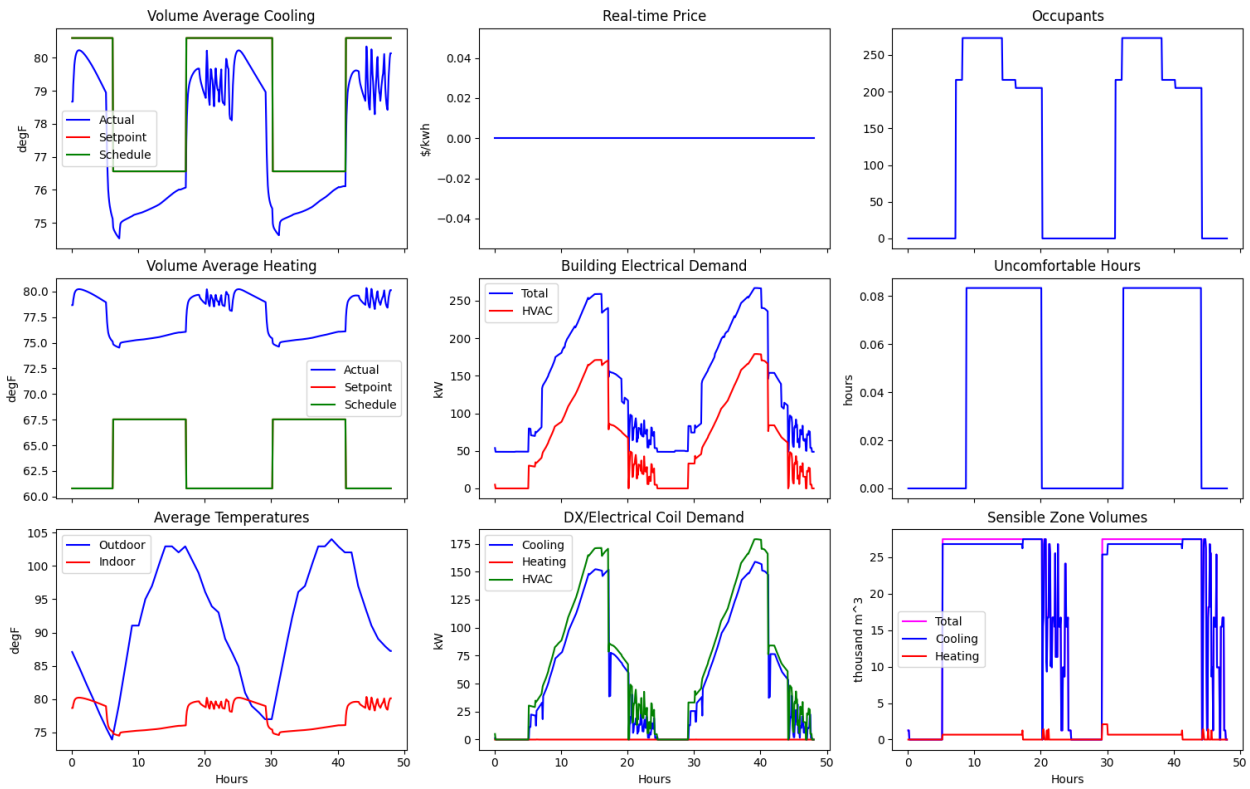


Transactive case

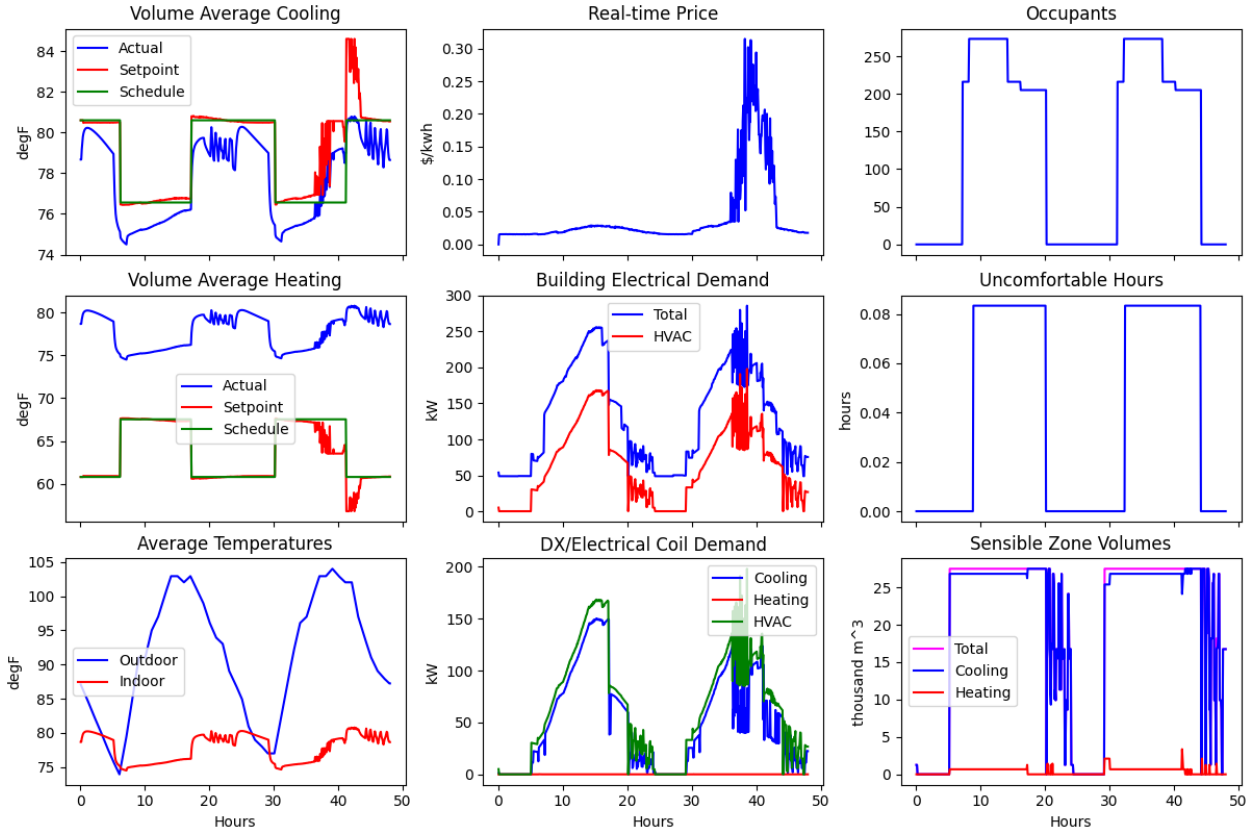


School (EnergyPlus)

Base case

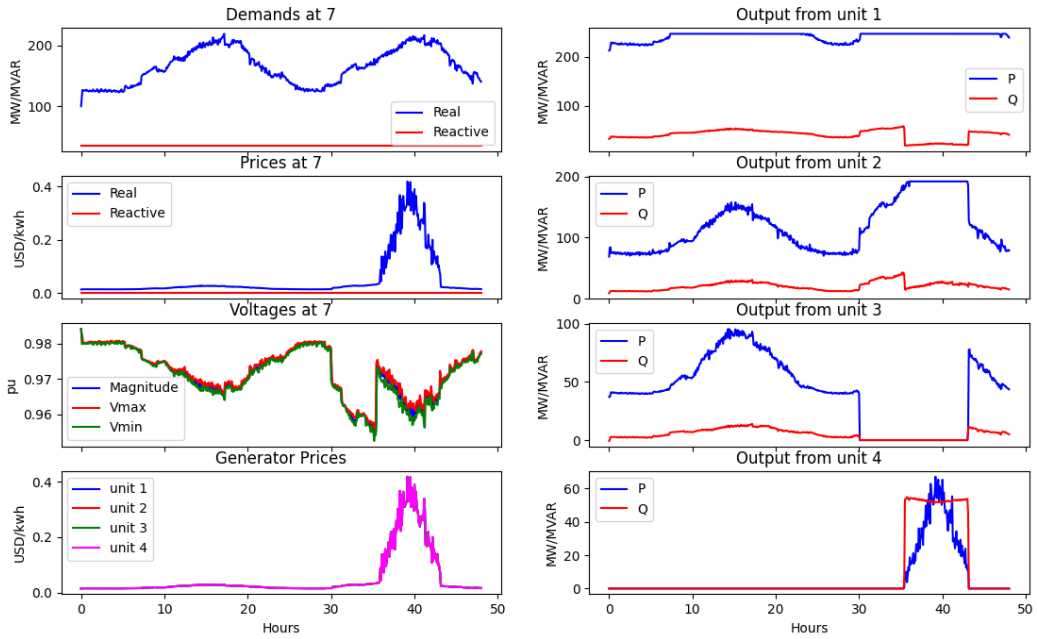


Transactive case

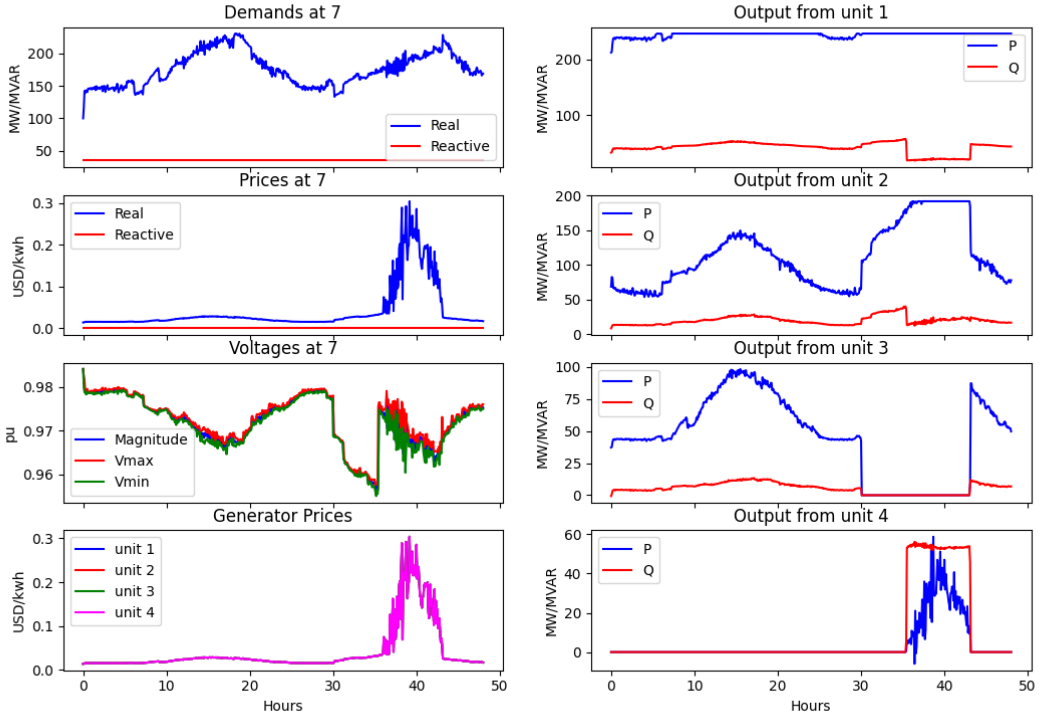


Bulk Power System Generators (PYPOWER)

Base case

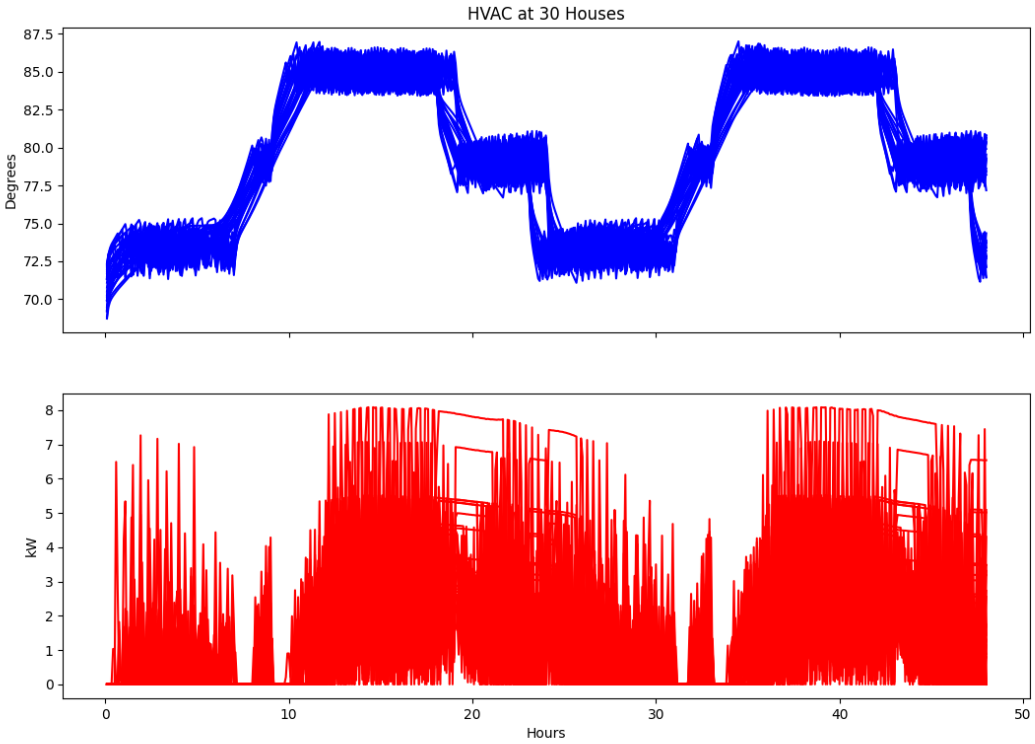


Transactive case

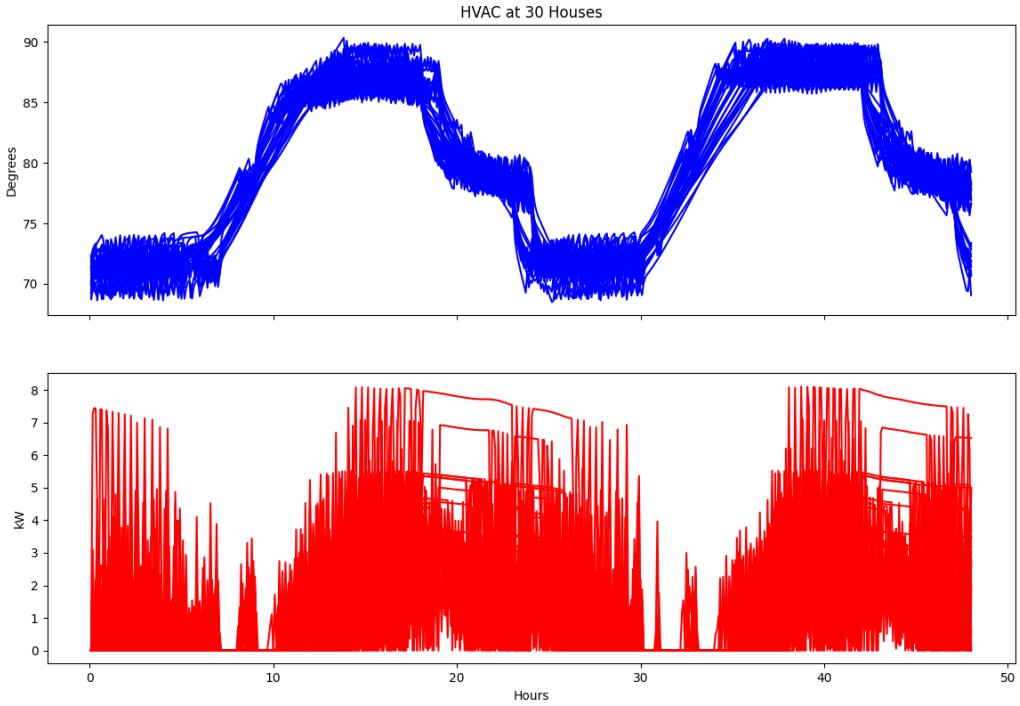


Residential HVAC (GridLAB-D)

Base case

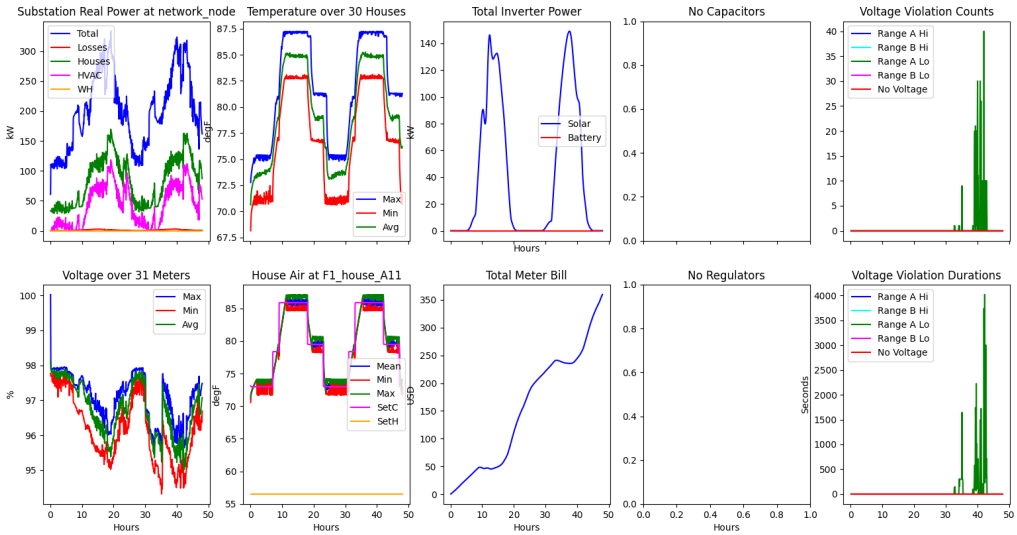


Transactive case

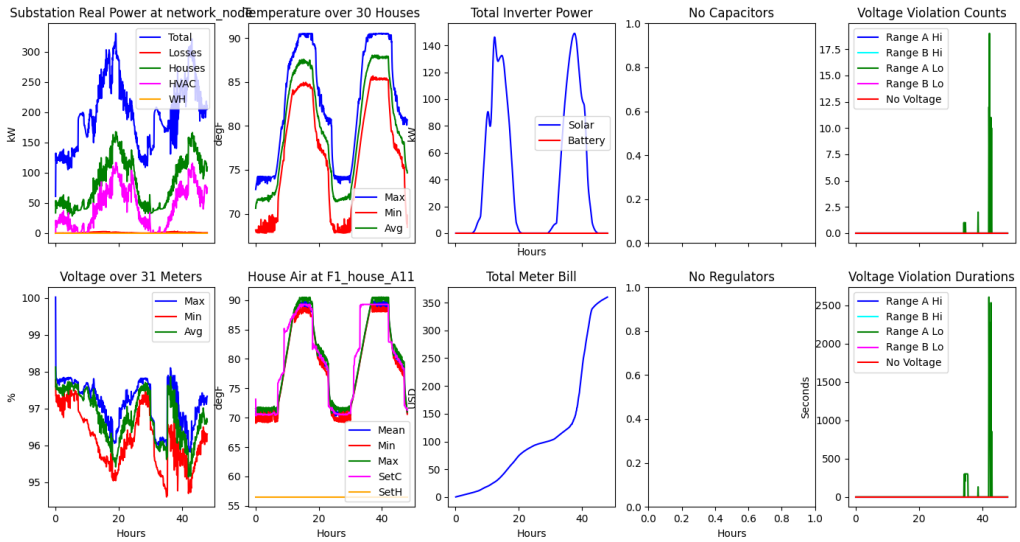


Rooftop Solar PV (GridLAB-D)

Base case

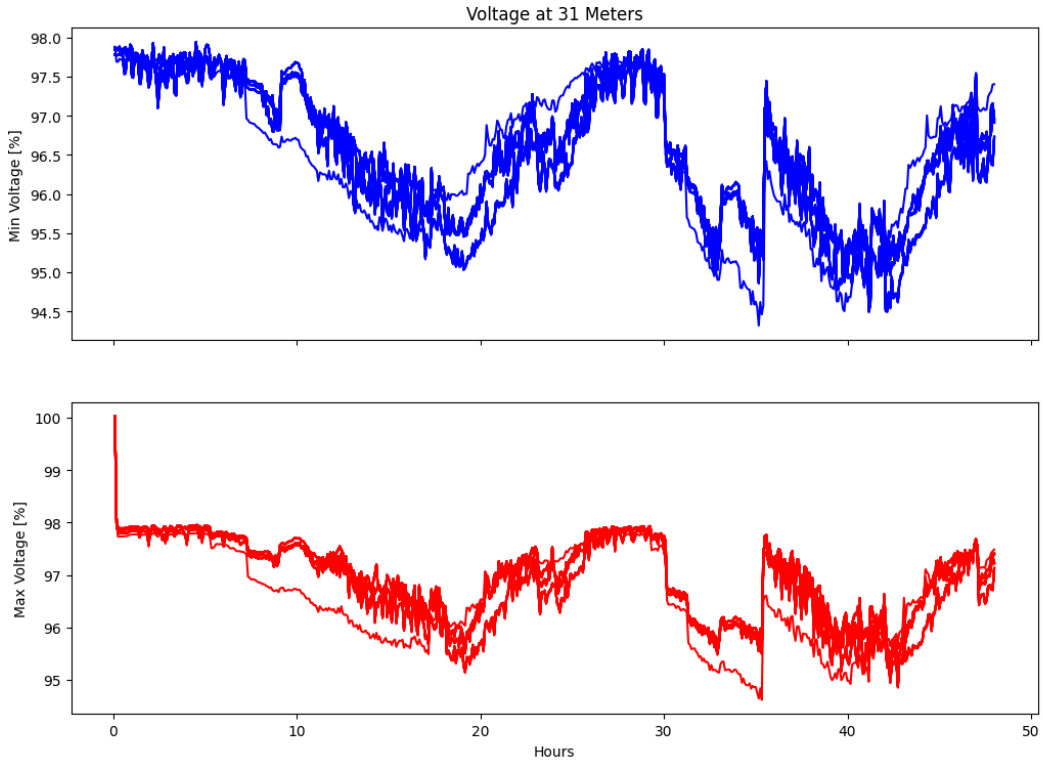


Transactive case

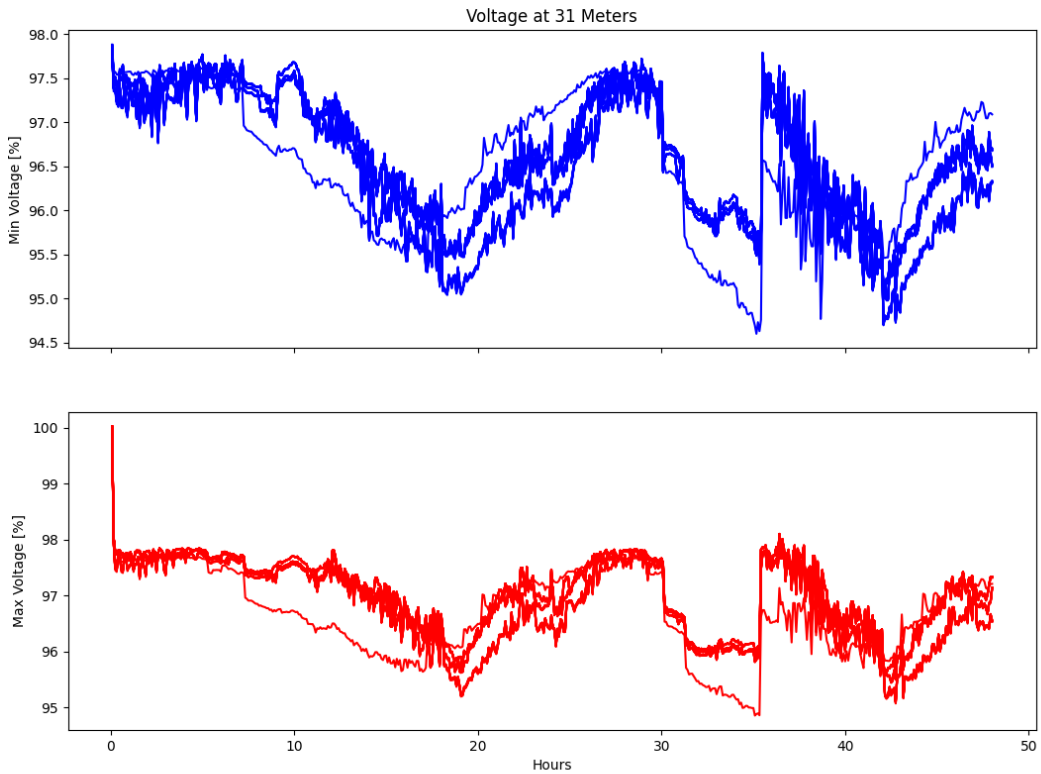


Residential Votlage (GridLAB-D)

Base case



Transactive case



File Listing

- *clean.sh* - script that removes output and temporary files
- *DeDeprecate.py*
- *eplus.json* - HELICS configuration file for EnergyPlus
- *eplus_agent.json* - HELICS configuration file for the EnergyPlus agent
- *monitor.json* - Monitor configuration file
- *NonGLDLoad.txt* - text file of non-responsive loads on transmission buses
- *outputs_te.glm* - defines GridLAB-D data to record
- *phase_A.player*
- *phase_B.player*
- *phase_C.player*
- *plots.py* - makes 5 pages of plots for a case: eg 'python plots.py TE_Challenge'
- *prepare_case.py* - sets up the dictionaries and GLD/Agent FNCS configurations for all cases
- *pypower.json* - HELICS configuration file for PYPOWER
- *README.rst* - this file

- *runh.sh* - script for the case with market
- *runh0.sh* - script for the case without transactive real-time energy market
- *te30_pp.json* - PYPOWER bulk power system model definition
- *TE_Challenge.glm* - GridLAB-D system definition
- *TE_Challenge_monitor.json* - HELICS configuration for monitor

3.1.7 DSO Stub

The dsostub example demonstrates TESP's ability to replicate the power system and market behavior of a distribution system without doing the detailed modeling that is often done for transactive studies such as in GridLAB-D. dsostub replicates the behavior in aggregate, allowing the computation burden in assessing transactive systems under development to be dramatically reduced.

dsostub supports a real-time and day-ahead double-auction energy market implementation. The bulk power system physics and market models are run by PSST with the CBC solver used for solving the security constrained unit commitment problem as part of clearing the day-ahead market. PSST with CBC also solves a security-constrained economic dispatch problem to clear the five-minute real-time energy market.

Almost of the parameters for the models used in this dsostub example are defined in the "case_config.json" file; this includes the transmission and generation system model. The "dsostub.py" script has a price-responsive curve hard-coded into the real-time market bidding.

This TESP capability was published in the IEEE Power and Energy Society General Meeting in July of 2021 and the publication can be found in [11] (https://doi.org/10.1109/PESGM46819.2021.9638030).

Running the demonstration

```
./runstub.sh "dsostub_case"
cd dsostub_case
./run.sh
```

File Directory:

- *case_config.json*: configuration data co-simulation including bulk-power system definition, source data file references, and general configuration and metadata
- *data*: folder containing time-series data used by various actors in the co-simulation
- *dsoStub.py*: Minimal representation of the distribution system providing identical interface as other TESP examples but requires dramatically reduced computation load as compared to full modeling traditionally done in GridLAB-D.
- ***runstub.sh*: prepares co-simulation "dstostub_case" directory with**
  - *case_config.json*: copy of the configuration data co-simulation
  - *[federates in the co-simulation].json*: related message diction
  - *run.sh*: run the co-simulation
  - *kill.sh*: shuts down the co-simulation

- *clean.sh*: clean the metric outputs a for another run
- *docker-run.sh*: example script to run a docker
- *monitor.sh*: example script to monitor a run and then start postprocessor
- *postprocess.sh*: example script to make and move a case directory

3.1.8 IEEE 8500

These example files are based on the IEEE 8500-node Feeder model, as adapted for the SGIP-3 use case and the NIST TE Challenge 2. This is a larger example and takes longer to run. More information is available at https://pages.nist.gov/TEChallenge/library/ and panel presentations from IEEE ISGT 2018. The backbone feeder model is documented at https://ieeexplore.ieee.org/document/5484381/

Running the base case

1. A current build of GridLAB-D from branch feature/1048 (or newer feature/1173) is required.
2. "runIEEE_8500.sh" run the base case. This example simulates one day but can take tens of minutes.

In order to plot results from the JSON files, Python 3 and the matplotlib package can be used:

1. "./runGLMDiction.sh" will create circuit metadata for plotting.
2. "plots IEEE_8500" will plot various metrics when the simulation finishes.

Alternatively, you can insert "recorders" into IEEE_8500.glm, which will create CSV files for plotting and post-processing. The simulation takes longer with CSV file output.

Notes on building or modifying the base case:

1. Weather CSV files were made from the adjust*.m files to create sunny and cloudy days from TMY data.
2. Feeder generator MATLAB scripts add houses, water heaters, air conditioners, solar panels and batteries to the 8500-node feeder base model in the backbone subdirectory. One produces IEEE_8500.glm for the base case, used by all teams. The other, found under PNNLteam subdirectory, produces inv8500.glm for PNNL simulations of smart inverters.
3. The house*.csv files contain equivalent thermal parameter (ETP) model parameters exported from GridLAB-D. These may be helpful if simulating houses on your own. See http://gridlab-d.shoutwiki.com/wiki/Residential_module_user%27s_guide for information about GridLAB-D's house model, including equivalent thermal parameters (ETP).

Base File Directory

- *adjust_solar_direct.m*: MATLAB helper function that ramps the direct solar insulation during a postulated cloud transient
- *adjust_temperature.m*: MATLAB helper function that ramps the temperature during a postulated cloud transient
- *backbone*: the IEEE 8500-node model as defined by the original authors for OpenDSS
- *CAISO_DAM_and_RTP_SG_LNODE13A_20170706-07_data.xlsx*: optional day-ahead market and real-time locational marginal price (LMP) data
- *clean.bat*: Windows batch file that removes output and temporary files
- *clean.sh*: Linux/Mac script that removes output and temporary files

- *climate.csv*: hourly temperature, humidity, solar_direct, solar_diffuse, pressure, wind_speed read by IEEE_8500.glm: **copy either sunny.csv or cloudy.csv to this file, depending on which case you wish to run**
- *Cloudy_Day.png*: screen shot of process_gld.py plots for the cloudy day
- *Cloudy_Voltages.png*: screen shot of process_voltages.py plots for the cloudy day
- *cloudy.csv*: copy to *climate.csv* to simulate a day with afternoon cloud transient
- *Cloudy.fig*: MATLAB plot of the correlated cloudy day temperature and solar irradiance
- *estimate_ac_size.m*: MATLAB helper function that estimates the house HVAC load as determined by GridLAB-D's autosizing feature. These values are embedded as comments in *IEEE_8500.glm*, which may be useful if modeling the HVAC load as a ZIP load.
- *gld_strict_name.m*: MATLAB helper function to ensure GridLAB-D names don't start with a number, as they can with OpenDSS files under *backbone*, but is not allowed in GridLAB-D
- *house_ca.csv*: house ETP parameters from the base case: may be useful for your own house models.
- *house_cm.csv*: house ETP parameters from the base case: may be useful for your own house models.
- *house_ua.csv*: house ETP parameters from the base case: may be useful for your own house models.
- *house_um.csv*: house ETP parameters from the base case: may be useful for your own house models.
- *IEEE_8500.glm*: base case feeder, populated with houses, PV and batteries.
- *IEEE_8500gener_whouses.m*: MATLAB script that produces *IEEE_8500.glm* from files under *backbone*
- *main_regulator.csv*: total feeder current in the base case: may be useful in benchmarking your own power flow solver.
- *PNNLteam*: subdirectory with PNNL's participation files: see next section.
- *README.md*: this file
- *schedules.glm*: cooling, heating, lighting and other end-use appliance schedules referenced by *IEEE_8500.glm*
- *substation_load.csv*: total feeder load and positive sequence voltage in the base case: may be useful in benchmarking your own power flow solver.
- *Sunny_Day.png*: screen shot of process_gld.py plots for the sunny day
- *Sunny_Voltages.png*: screen shot of process_voltages.py plots for the sunny day
- *sunny.csv*: copy to *climate.csv* to simulate a clear, sunny day
- *sunny.fig*: MATLAB plot of the correlated sunny day temperature and solar irradiance
- *TE_Challenge_Metrics.docx*: documentation of the use case metrics of interest to the entire NIST TE Challenge 2 team

PNNL Team Files

The subdirectory *PNNLteam* contains files used only for the pre-cooling thermostat and smart inverter simulations, as presented by PNNL at IEEE ISGT 2018. See the report on NIST TE Challenge 2 for more details. To run these simulations, you will need to install TESP, which includes FNCS and the *tesp_support* Python package. These simulations require a recent build of GridLAB-D from the feature/1173 branch (newer than the version posted for the base case), which is included with TESP. Also, newer versions of TESP run on Linux only. For Windows or Mac OS X, you will have to run TESP in a virtual machine or Docker container.

inv30.glm is a small 30-house test case with smart inverters, and *inv8500.glm* is the larger feeder model with smart inverters.

invti30.glm is the 30-house test case with smart inverters, and the house *thermal_integrity_level* attribute specified instead of the individual R values and airchange_per_hour values. The log file *precoolti30.log* will contain the house equivalent thermal parameter (ETP) model as estimate from the thermal integrity level.

All three run over FNCS with the precooling agent in tesp_support.precool. Since ISGT 2018, some changes have been made to the precooling agent:

- Only 25 houses are allowed to change setpoint at each time step: others wait until a subsequent step
- The precooling temperature offset is randomized from 1.9 to 2.1 degrees

These simulations take up to 1 hour to run. Example steps are:

a. "python3 prepare_cases.py"
b. "./run8500.sh"
c. "python3 plots.py inv8500" after the simulation completes
d. "python3 bill.py inv8500"
e. "python3 plot_invs.py inv8500"

There are three GridLAB-D definitions near the top of *inv30.glm*, *invti30.glm* and *inv8500.glm*. These determine the solar inverter control modes, and (only) one of them should be enabled. The script files do this on the command line to GridLAB-D, e.g., *-D INV_MODE=VOLT_VAR*. Inside the GLM files, one and only one of the following lines must be left uncommented:

- #define INVERTER_MODE=${INV_MODE}
- //#define INVERTER_MODE=CONSTANT_PF
- //#define INVERTER_MODE=VOLT_VAR
- //#define INVERTER_MODE=VOLT_WATT

InvFeederGen.m was adapted from *IEEE_8500gener_whouses.m* in the parent directory, to populate *inv8500.glm* in a similar way, but with smart inverter functions added. See the TESP documentation for guidance on interpreting the other files in this directory.

- *bill.py*: calculates and plots a summary of meter bills
- *clean.sh*: script to clean out log files and output files
- *inv30.glm*: a 30-house test case with smart inverters
- *inv8500.glm*: the 8500-node test case with smart inverters
- *invti30.glm*: a 30-house test case with smart inverters and simplified house thermal integrity inputs
- *invFeederGen.m*: a MATLAB helper script that populates 8500-node with smart inverters, based on the ../backbone directory
- *kill5570.sh*: helper script that stops processes listening on port 5570
- *parser.py*: testing script for parsing FNCS values
- *plot_invs.py*: tabulates and plots the meter with most overvoltage counts
- *plots.py*: plots the GridLAB-D and agent outputs using tesp_support functions
- *prepare_cases.py*: prepares the JSON dictionaries and FNCS configuration for both cases, using tesp_support functions
- *prices.player*: time-of-day rates to publish over FNCS
- *run30.sh*: script that runs the 30-house case, inverters in constant power factor mode

- *runti30.sh*: script that runs the 30-house case with simplified thermal integrity input, and volt-var mode inverters
- *run8500.sh*: script that runs the 8500-node case with no price, voltage or smart inverter response
- *run8500base.sh*: script that runs the 8500-node case, responsive to time-of-use rates and overvoltages
- *run8500tou.sh*: script that runs the 8500-node case, price response to time-of-use rates, no smart inverters
- *run8500volt.sh*: script that runs the 8500-node case, precooling response to overvoltage, no smart inverters
- *run8500vvar.sh*: script that runs the 8500-node case, non-transactive, smart inverter volt-var mode
- *run8500vwatt.sh*: script that runs the 8500-node case, non-transactive, smart inverter volt-watt mode

3.1.9 House Example

This example populates houses, solar and storage onto an existing GridLAB-D feeder, using the *write_node_houses* and *write_node_house_config* functions from *tesp_support*. It does not run a transactive system.

To run the example:

```
./run.sh
#Use *ps* to learn when GridLAB-D finishes. It should take a couple of minutes.
python3 plots.py
```

File Directory

- *clean.sh:* script that removes output and temporary files
- *gld_plots.png:* results for the feeder, house load exceeds substation load because of solar generation
- *hvac_plots.png:* results for the house air conditioning systems
- *meter_plots.png:* results for the billing meters
- *plots.py:* makes 3 pages of plots
- *README.rst:* this file
- *run.sh:* script that writes houses, creates a dictionary and runs GridLAB-D
- *test_houses.glm:* GridLAB-D file that includes *houses.glm* and runs for two days
- *WriteHouses.py:* writes houses on 14 primary nodes to *houses.glm*

Results

The example feeder has 14 primary nodes, populated as follows. 80% of the houses have air conditioning. Except where indicated, all are in region 2. All use a fixed seed for randomization and use a metrics collection interval of 300. See *WriteHouses.py* for details on how to set up the populations, and function documentation for other options.

Table 3.1: Model description

Node	Phasing	kVA	Drop[ft]	Region	Houses	Other
F7B1	ABC	1000	0	2	42	
F7B2	AS	500	0	2	42	
F7B3	BS	500	0	2	42	
F7B4	CS	500	0	2	42	
F7B5	AS	500	75	2	42	triplex drop
F7B6	BS	500	75	2	42	triplex drop
F7B7	CS	500	75	2	42	solar and storage, triplex drop
F7B8	ABC	1000	0	2	40	uses loadkw and house_avg_kw
F1B1	ABC	1000	0	1	42	
F1B2	ABC	1000	0	3	42	
F1B3	ABC	1000	0	4	42	
F1B4	ABC	1000	0	5	42	
F1B5	ABC	1000	75	2	42	quadriplex service drop
F1B6	ABC	1000	0	2	42	solar and storage

Plotted results from this example.

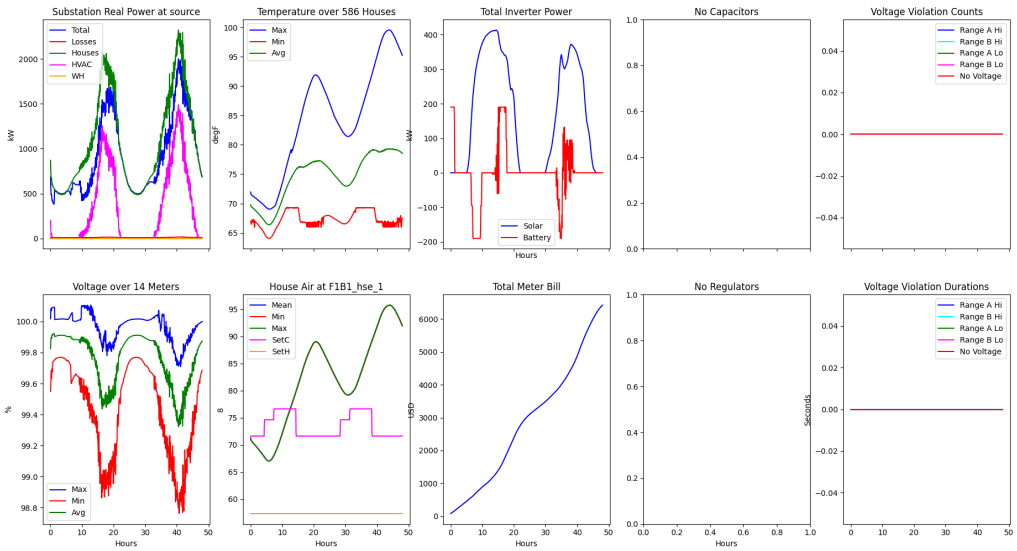


Fig. 3.4: Results collected from GridLAB-D measurements

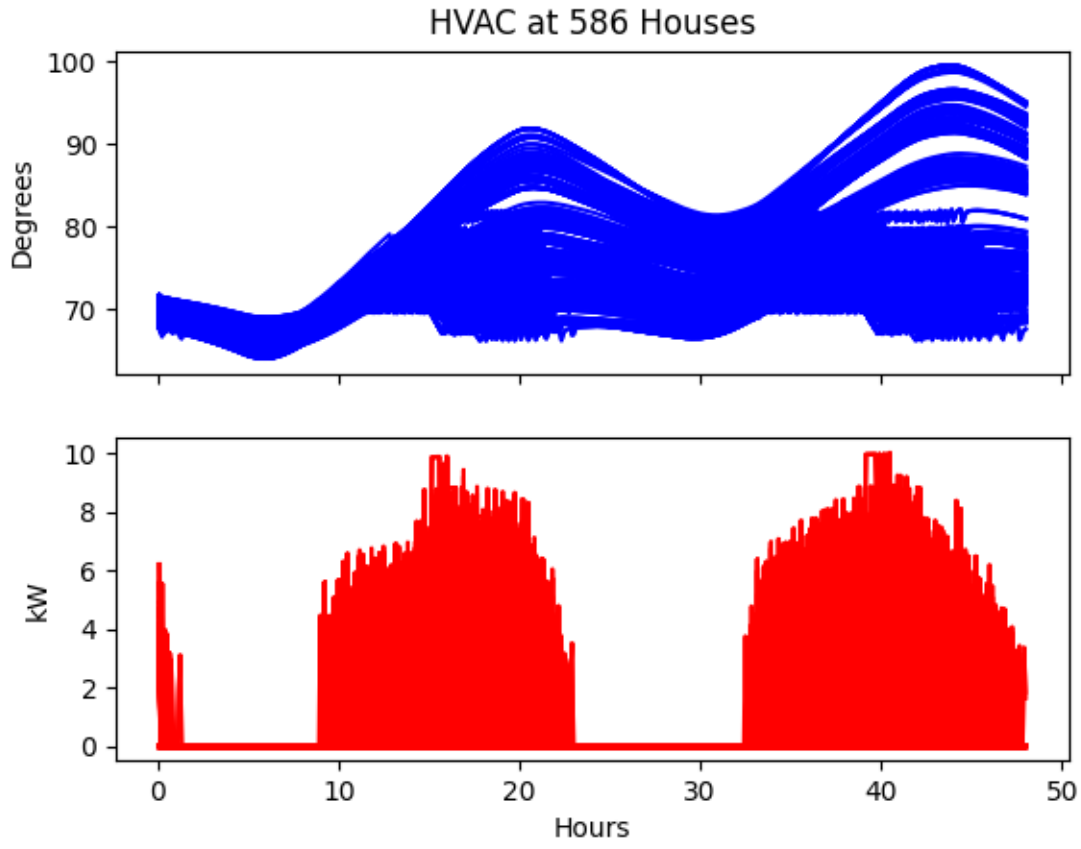


Fig. 3.5: Operation of all HVACs in GridLAB-D

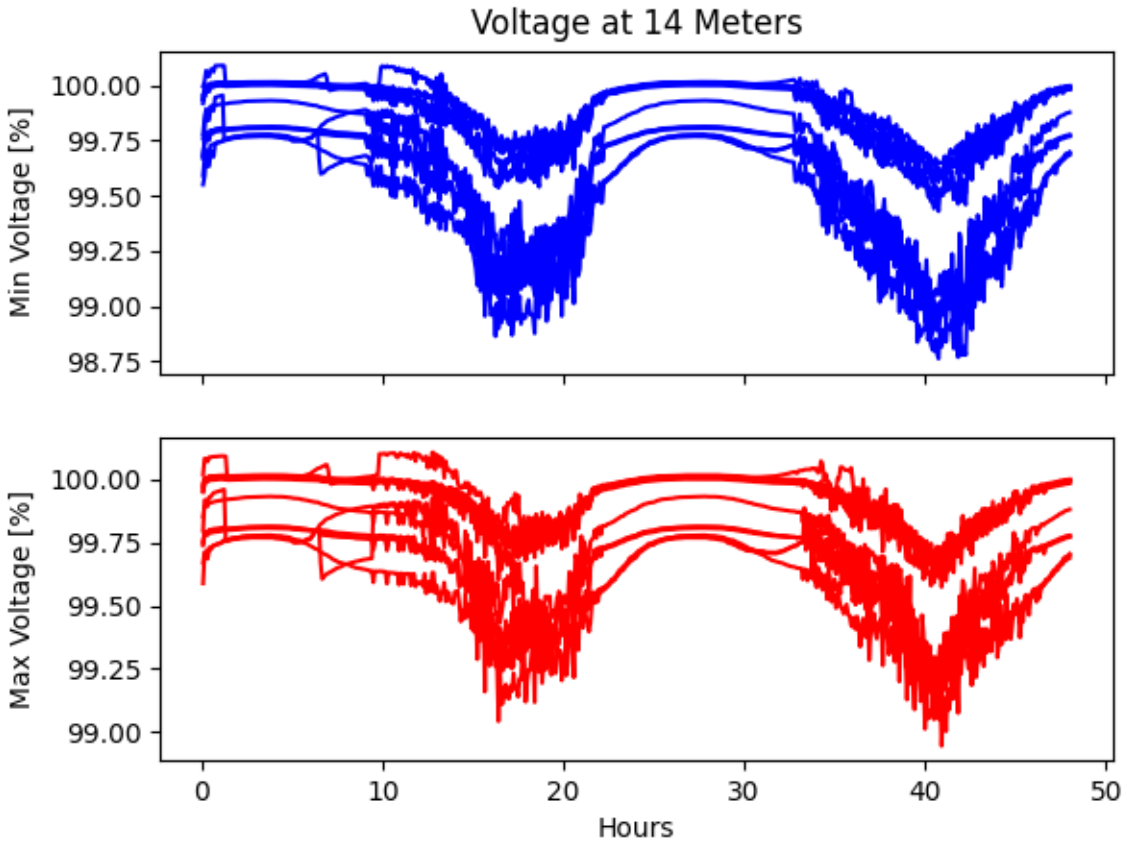


Fig. 3.6: Meter measurements of all houses in GridLAB-D

3.1.10 GridLAB-D Player and Recorder Demonstration

This example demonstrates how to use GridLAB-D's player and recorder functions. GridLAB-D's player functionality allows external data to be played into a simulation and define the value of any of the parameter in most objects. Recorders do the converse and record the value of any object's parameter and writes it out to file. Further details can be found in GridLAB-D's documentation on players and recorders.

Directories TE30 and ieee8500 provides more in-depth examples.

To run the example:

1. ./run.sh

File Directory

- *README.rst*: this file
- *run.sh*: script that writes houses, creates a dictionary and runs GridLAB-D
- *SG_LNODE13A.player*: player file that has lmps values for a node for testing
- *testImp.glm*: GridLAB-D file that an SG_LNODE13A.player that uses the object player

Results

The file lmp_value.csv is created and can be compared with SG_LNODE13A.player which was used for input.

3.1.11 GridLAB-D Model Modification

For many transactive energy analysis, detailed modeling of the distribution system is an important part of the analysis and GridLAB-D is often chosen as the modeling tool. It is not unusual, though, for a given GridLAB-D model to be non-ideal for the particular analysis scenario to be modeled and need modification in some way: adding rooftop solar, replacing the existing heat pumps with higher efficiency models, changing voltage regulator settings, etc. To aid in this, TESP includes a GridLAB-D modification API to provides a class- and object-oriented way of manipulating the model that we hope provides a (relatively) painless process for editing GridLAB-D models.

GLMModifier Philosophy

There have been several other scripts previously developed that provide GridLAB-D model modification. Most of these are text-line-based where the model is read in line-by-line and at appropriate points lines are editing before being printed out to file or additional lines are inserted before moving through the model (for example, adding rooftop solar). Though these have been used for over a decade they always presented a challenge in that model modifications could not be made holistically as the entire model was not parsed but rather remained as text that could be manipulated.

TESP's GLMModifier overcomes these shortcomings by providing an internal data structure that the model can be read and parsed into. By doing so, the modeler has the ability to evaluate the entire model, manipulate the necessary portions, and then write out the entire model to file. For example, after GLMModifier reads in the model parses it into its data structure, it is possible to count the number of houses that use gas heating, convert them to use heat pumps, and upgrade any necessary transformers and power lines to handle the increased load.

There are two primary ways to access the model using GLMModifier:

- Object- or class-based - GLMModifier's data structure allows for evaluation and modification of GridLAB-D objects based on their class. For example, all GridLAB-D "house" objects are easily accessible and their parameters editable.

- Graph-based - GLMModifier uses the networkx library to create a graph of the electrical network. Using this library its possible to evaluate the GridLAB-D model in terms of electrical connectivity and modify in more specific ways. For example networkx allows the modeler to identify which components lie between a newly added load and the substation so their capacity can be evaluated and potentially increased.

GLMModifier API Example Walk-Through

The following is a walk-through of an example use of the GLMModifier API ("gld_modifier.py") in the TESP "examples/capabilities" folder.

GLMModifier is Python-based so the first step (after installing TESP) is importing the appropriate libraries.:

```
from tesp_support.api.modifier import GLMModifier
from tesp_support.api.data import feeders_path
```

Then create a GLMModifier object to do the manipulation:

```
glmMod = GLMModifier()
```

With the modifier, we can read the GridLAB-D model into the GLMModifier data structure for manipulation:

```
glmMod.model.read('path/to/model.glm')
```

GridLAB-D split their functionality into various modules and for this example, we're going to be adding houses to the model which means we need to make sure the "residential" module gets added to the model file.:

```
glmMod.add_module('residential', [])
```

The GLMModifier has an attribute that holds the entire GridLAB-D model, "glm". Purely to make our live a few characters easier when using the GLMModifier, we can assign this to another variable of our choice and strip out the need constantly pre-pend many of our commands with "GLMModifier".:

```
glm = GLMMod.glm
```

GLMModifier makes it easy to get the names of all of the objects of a given class and in this case, to add houses, we need to look for GridLAB-D's "triplex_meters" to attach the houses to.:

```
tp_meter_objs = glm.triplex_meter
tp_meter_names = list(tp_meter_objs.keys())
```

`tp_meter_objs` is a Python dictionary with the keys being the object names and the value being an Python dictionary of the object parameters and values. To make a list of the names of the meters, we just need to ask for the keys of the dictionary as a list.

Adding Objects

`tp_meter_names` is a list of the names of the GridLAB-D `triplex_meter` objects as strings. Using those names we can build up a Python dictionary that defines the parameters of another `triplex_meter` object we're going to add to the model. The dictionary is called "meter_params" and has three members all defined by the data from an existing specific `triplex_meter` in the model.:

```
new_name = tp_meter_names[house_num]
billing_meter_name = f"{new_name}_billing"
```

(continues on next page)

(continued from previous page)

```
meter_params = {
    "parent": new_name,
    "phases": glm.triplex_meter[f"{new_name}"]["phases"],
    "nominal_voltage": glm.triplex_meter[f"{new_name}"]["nominal_voltage"],
}
```

The `phases` and `nominal_voltage` are easily defined using GLMModifier as they are just members of a dictionary that defines a specific triplex meter.

Once the Python dictionary with the GridLAB-D object parameters are defined, it can simply be added to the model.:

```
glmMod.add_object('triplex_meter', billing_meter_name, meter_params)
```

`.add_objects()` has three parameters: the type of object, the name of the object (which the API uses to define the `name` parameter of the object behind the scenes), and the dictionary with the object parameters.

Adding and Modifying Existing Object Parameter Values

Further down in the example, there's a portion of code showing of how to modify an existing object. In this case, we use the fact that `.add_object()` method returns the the GridLAB-D object (effectively a Python dictionary) once it is added to the model. Once you have the GridLAB-D object, its easy to modify any of its properties such as:

```
house_obj['floor_area'] = 2469
```

This exact syntax is also valid for adding a parameter that is undefined to an existing GridLAB-D object.

Deleting Existing Object Parameter Values

To delete a GridLAB-D object parameter value, you can just set to to *None*:

```
house_to_edit["Rroof"] = None
```

Note that GridLAB-D requires some parameters to be defined to run its simulations. Removing the parameter will remove it from the GridLAB-D model file that gets created (.glm) but may effectively force GridLAB-D to use its internal default value. That is, clearing the parameter value in this way is not the same as setting it to an undefined value.

Deleting Existing Objects

Its possible to delete an object and all its parameter values from the GridLAB-D model:

```
glmMod.del_object('house', house_to_delete)
```

To prevent problems with electrical continuity of the models, by default this method will delete children objects.

networkx APIs

networkx library is a general graph Python library and it utilized by TESP to store the topology of the electrical network in GridLAB-D. The core GLMModifier APIs are oriented around the GridLAB-D classes and their objects in the model and from these the topology of the electrical circuit can be derived but not easily or quickly. To make topology-based modifications easier, we've done the hard work of parsing the model and building the networkx graph. With this graph, modelers can more easily and comprehensively explore and edit the model.

First, if any edits have been made to the GridLAB-D model since importing it, the networkx object needs to be updated prior to include those changes. Conveniently, this also returns the networkx graph object:

```
graph = glmMod.model.draw_network()
```

As you can see, the networkx graph is a property of the GLMModifier.model object and the above line of code simply makes a more succinct reference to it.

After that, you can use networks APIs to explore the model. For example, starting at a particular node, traverse the graph in a breadth-first manner:

```
for edge in nx.bfs_edges(graph, "starting bus name"):
```

For each edge you, the modeler, can look at the properties of each edge (GridLAB-D link objects) to see if it is of particular interest and modify it in a specific way.

Plotting Model

GLMModifier includes the capability of creating a visual representation of the network for manual inspection. This allows the user to evaluate the model and make sure the changes made are as expected and has the topology expected. To create the plot of the graph of the model a simple API is used:

```
glmMod.model.plot_model()
```

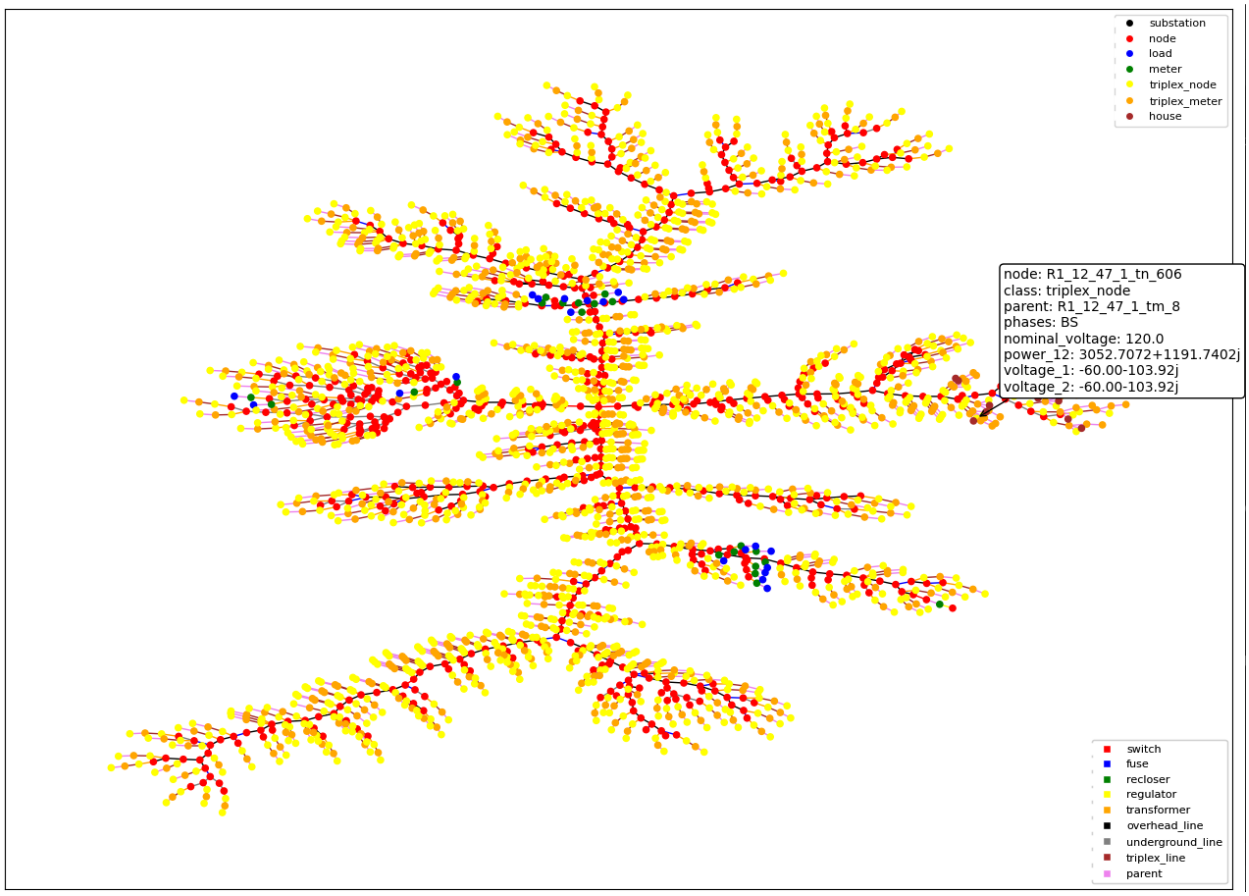
Under the hood, this API makes an update to the networkx graph and then automatically lays it out and plots it on screen, as shown below.

Mousing over the nodes of the system shows some of the metadata associated with them; in the example image shown above one of the houses is selected. As of this writing, this metadata is not available for the links/edges in the graph but we're anticipating adding that data soon. The layout chosen is algorithmic and does not respect coordinates that may be present in the imported .glm. For larger networks, it can take tens (or many tens) of seconds for the layout to complete; creating the graph is a blocking call in the script and the rest of the script will not run until the plotting window is closed.

Writing Out Final Model

Once all the edits to the model have been made, the model can be written out to file as a .glm and run in GridLAB-D.:

```
glmMod.write_model("output file path including file name.glm")
```


substation
node
load
meter
triplex_node
triplex_meter
house
node: R1_12_47_1_tn_606
class: triplex_node
parent: R1_12_47_1_tm_8
phases: BS
nominal_voltage: 120.0
power_12: 3052.7072+1191.7402j
voltage_1: -60.00-103.92j
voltage_2: -60.00-103.92j
switch
fuse
recloser
regulator
transformer
overhead_line
underground_line
triplex_line
parent

GLMModifier House Object Population

Previous GridLAB-D model modification tools also included methods by which to choose the parameters for some objects (the house object in particular). The re-implementation of these features using updated data and methodologies are currently being implemented in what we are calling a "reference implementation" to show others one possible way of defining values for all these parameters. We want to not only provide an empirically-based method but also clearly document it so that other users can better understand what we did and customize or modify it to better suit their needs.

Future work

We've put in a lot of work to support all of GridLAB-D syntax but are not quite there yet. In particular, the last remaining element we haven't been able to capture well in our data structure are the `#ifdef` C-like conditionals GridLAB-D supports. This feature is under active development.

Currently, when GLMModifier writes out the model it does so in a manner that groups all the classes together. Alternative methods of writing out this non-linear data structure need to be evaluated so that human-readers of the file have an easier time (at least in some cases). This is on our to-do list.

3.1.12 Datastore Demonstration

Introduction

The TESP data store provides a way to collect simulation data (input and output data sets) for use outside the compute space in which it was created. For example, a particular transactive study may require above-average computational resources and thus is run on a dedicated server. Once the simulation is complete, the TESP datastore capability can be used to collect the relavant results and then handed off to an analyst to perform the post-processing. Using the datastore allows the relevant files to be extracted from the place they were created in the file system and distributed to the end user in consistent file structure. This allows the paths to the relevant files to be fixed and allow post-processing to be developed and run by any user that is given the data store.

The remainder of this page will describe who to use the datastore capability in TESP, both for getting data into the store and pulling it out to use. We'll be using data produced by the TE30 example as it can be run relatively quickly but has a diverse set of data that can be used.

Storing Data

The TESP datastore has two parts:

- A ZIP file with all raw data collected
- A metadata JSON that shows the contents of each file

The construction of these two files for the store are done independently but are obviously related. Though there are exceptions, it is likely that most of the time you will want to both add a file to the ZIP as well as add metadata about that file to the JSON. Most users of the datastore you make will want access to both the data itself as well as the metadata so they can better understand the data and how best to use it.

Adding Files to the ZIP

The TESP datastore API provides two general strategies for adding files to the ZIP:

- Add a directory to the store and recurse down through the file hierarchy adding all files and sub-directories to the ZIP
- Add individual files

Both approaches allow some degree of locating files in the ZIP file/folder hierarchy differently than in the file/folder hierarchy from which they come. When adding individual files this freedom is absolute; the source path for the data and the destination path are independently defined. When adding an entire folder (and its contents) only the new root location for this folder can be defined; the file/folder hierarchy beneath this is preserved.

Adding Metadata to JSON

The TESP datastore API can only store data that is tabular in nature (as of this writing). Generally the simulators/tools that produce data in TESP are able to satisfy this requirement and given the time-series nature of transactive system analysis supported by TESP, it is likely that any custom tools or code could be made to produce similar tabular data.

The TESP datastore API is able to automatically find the tables in the data file. The "table" concept is native to the HDF5 and SQL-style database files. For comma-separated-value (CSV) files, it is assumed all the data present is in one table (with perhaps some header lines). Once the tables are found, the API allows the person adding files to the metadata to define the location in the data of the timestamp data and the labels for the columns. This data is added to the metadata JSON to allow the user of the datastore to inspect the metadata as a guide or aid in performing the post-processing

Quick Example

The TESP installation includes a full working example with data from the TE30 example (discussed more below) but here's a quick and dirty overview of the APIs in action:

```
import tesp_support.api.store as store

store = store.Store("*file name of store*") # Create store
os_data_path = store.add_path("*OS path to data*", "*name of path for metadata*")
zip_dir_path = os_data_path.set_includeDir("*ZIP path to data*", *recurse?[bool]*)

# Adding file to metadata JSON
data_file = store.add_file("*OS file path + name relative to current working dir*",
                               "*name in store*",
                               "*description string for data*")
table_list = data_file.get_tables()
# Just doing the first table to keep it simple
data_file.get_columns(table_list[0])
data_file.set_date_bycol(table_list[0], "*column name with timestamp information*")

# Adding files to ZIP in previously created directory in ZIP
store.set_includeFile(zip_dir_path, "*OS path to file to include*")
```

Retrieving and Using Data

The datastore is intended to allow portability of data to allow post-processing scripts the ability to work from data in a known location (that is, relative to the datastore), inspecting the schema of the stored data, and providing data import and conversion into pandas DataFrames for ease of processing.

Quick Example

The essential APIs for using the data store are pretty simple.::

import tesp_support.api.store as store store = store.Store(*store name*) schema = store.get_schema(*data filename*) tables = schema.tables columns = schema.columns df = schema.get_series_data(*table name*, *start time*, *stop time*)

Example - TE30

The example provided with TESP uses the TE30 example as it runs fairly quickly (minutes instead of hours) and produces a diversity of data.

The use of the datastore is always split into two parts: creating the datastore from results files and using the datastore to analyze results. In this example, to save time, we've run the TE30 simulation for you and created the datastore already. After running the simulation (when the results files were created, filled with data, and finalized), the "makestore.py" script in the TE30 example folder was run and it created two files: "te30_store.json" and "te30_store.zip". These files can then be moved around and then a post-processing script, like "te30_ustestore.py" can be used to access and manipulate the data.

makestore.py

To emulate the case where the TE30 example is run on a different computer than the post-processing takes place, "makestore.py" resides in the "examples/capabilities/te30" folder and should be run after the TE30 case has been run and the output data is produced.

First, we create the datastore and add a directory to the metadata and .zip.:

```
my_store = fle.Store(case_name)
...
my_path = my_store.add_path("../te30", "TE30 Directory")
sub = my_path.set_includeDir(".", False)
```

The data being ingested by the store is from GridLAB-D and is in HDF5 format. Due to the way the data collection in GridLAB-D using HDF5 in the TE30 example is implemented, a number of results files are created with many of them being effectively empty. This is due to the fact that none of those object exist in the GridLAB-D model but results files are generated by GridLAB-D regardless of which types of objects exist in the GridLAB-D model. In this example we're just looking at the data collected from the billing meters, houses, inverters.

For each of the GridLAB-D data files being added, add it to the metadata JSON and to the ZIP. Again, particular to the way GridLAB-D records its data using HDF5, for each simulated day GridLAB-D generates a new table in the HDF5 file. Thus, the files is added to the ZIP only once but the metadata for each day is added to the JSON separately.:

```
my_file = my_store.add_file(name, name[i], names[i] + ' for ' + challenge)
my_path.set_includeFile(sub, name)
tables = my_file.get_tables()
if len(tables) > 1:
    columns = my_file.get_columns(tables[1])
```

(continues on next page)

(continued from previous page)

```
    my_file.set_date_bycol(tables[1], 'date')
    columns = my_file.get_columns(tables[2])
    my_file.set_date_bycol(tables[2], 'date')
```

As mentioned previously, the TESP datastore API assumes all data in a CSV is effectively a single table and thus is added singly. The second parameter in the .get_columns() method is optional but is particularly useful for CSVs that have header lines at the top of the file (such as these used in this example). The second parameter is the number of header lines to skip before getting to the row that defines the names of the columns.:

```
my_file = my_store.add_file(challenge + ".csv", challenge,'CSV for TE_ChallengeH')
tables = my_file.get_tables()
if len(tables):
    columns = my_file.get_columns(tables[0], 0)
    my_file.set_date_bycol(tables[0], 't[s]')
my_path.set_includeFile(sub, challenge + ".csv")
```

For completeness sake, a number of JSONs with simulation metadata are included in the ZIP but NOT cataloged in the metadata JSON. Including this simulation metadata will be useful for those post-processing the results but as it is not time-series, it is not cataloged in the datastore metadata JSON.:

```
names = ['agent_dict', '_glm_dict', 'model_dict']
for i in range(len(names)):
    my_path.set_includeFile(sub, challenge + names[i] + ".json")
```

te30_usestore.py

To run the "te30_usestore.py" example, first copy "te30_store.zip" to the "te30_store" folder. This is emulating somebody handing over data they produced by running TESP on another machine for you to post-process. Open up the .zip, allowing it to expand and giving you access to the datafile the user of "makestore.py" zipped up.

Once setting up the post-processing to define the time-range we're going to analyze and potentially change the working directory to that of the folder containing the unzipped files, we first create a datastore object.:

```
te30_store = store.Store(case_name)
```

If you don't have a good sense of the data being passed to you in the .zip, you can get a list of the schemas in the datastore.:

```
for item in te30_store.get_schema():
print(f"\t{item}")
```

Note that this is not necessarily a list of the files themselves. This is a list of the datafiles that have a defined schema in the datastore metadata. There could be other files that have not had schemas created but have been distributed in the zip.

To look at the specific data in a schema, we can make a call to get the schema and then look at the tables in a schema and the columns associated with each table.:

```
weather_schema = te30_store.get_schema("weather")
# If you're not evaluating the schema interactively you can print it to console
print(f"Weather tables {pp.pformat(weather_schema.tables)}")
print(f"Weather columns {pp.pformat(weather_schema.columns)}")
```

(continues on next page)

(continued from previous page)

```
...
inverter_schema = te30_store.get_schema("inverter_TE_ChallengeH_metrics")
```

Then we pull the data out.:

```
weather_data = weather_schema.get_series_data("weather", start_date_1, end_date_1)
...
inverter_data = inverter_schema.get_series_data("index1", start_date_1, end_date_1)
```

TESP uses pandas DataFrames as the standard format for time-series data, regardless of the source data format (*e.g.* .csv, .h5); the conversion is handled by the TESP APIs. In this case, the weather data was from a .csv and the inverter data was from an HDF5 file generated by GridLAB-D. When properly indexed so the timestamps for the data are recognized as such, pandas takes care of aligning the data in time so actions like plotting are much easier.:

```
weather_data = weather_data.set_index("timestamp")
inverter_data = inverter_data.set_index("date")
```

Unfortunately, the GridLAB-D data contains all inverter data in the same table and since we're going to look at the output of a single inverter, we have to filter the table to only show us the data for that inverter.:

```
houseA11_inv = inverter_data.loc[(inverter_data["name"] == b"inv_F1_house_A11")]
inverter_time = houseA11_inv["date"]
# If the date is typed as a string instead of a datetime object, we need to
# convert it to a datetime object to allow for indexing.
if isinstance(inverter_time.iloc[0], str):
    inverter_time = inverter_time.str[:-4]
    inverter_time = pd.to_datetime(inverter_time, format="%Y-%m-%d %H:%M:%S")
# Making a new DataFrame for convenience
inverter_data = pd.concat([inverter_time.T, houseA11_inv["real_power_avg"]], axis=1)
```

For this example, we're going to be confirming that the simulation shows a strong correlation between the solar flux (as recorded in the weather data) and the rooftop solar production power (as recorded by GridLAB-D). If all the models and the co-simulation are working right (and they better be because we're using it as an example for TESP), then we would expect good agreement between the two time-series datasets.

Once we've got the data from the two sources as DataFrames, the rest is just using pandas and Matplotlib to make our graph and visually evaluate the data as shown in Fig. 3.7.

As expected, a strong correlation does exist and all is well.

TESP uses TCP/IP port 5570 for communication and requires Python 3. Simulations can start many processes, and take minutes or hours to complete. At this time, instructions are given only for the Linux package or Docker version of TESP, which is installable. See below for advice on running TESP in native Mac OS X or Windows.

Some general tips for Linux:

- invoke **python3** instead of just *python*
- we recommend 16 GB of memory
- high-performance computing (HPC) should be considered if simulating more than one substation-week
- you may have to increase the number of processes and open file handles allowed
- **lsof -i :5570** will show all processes connected to port 5570

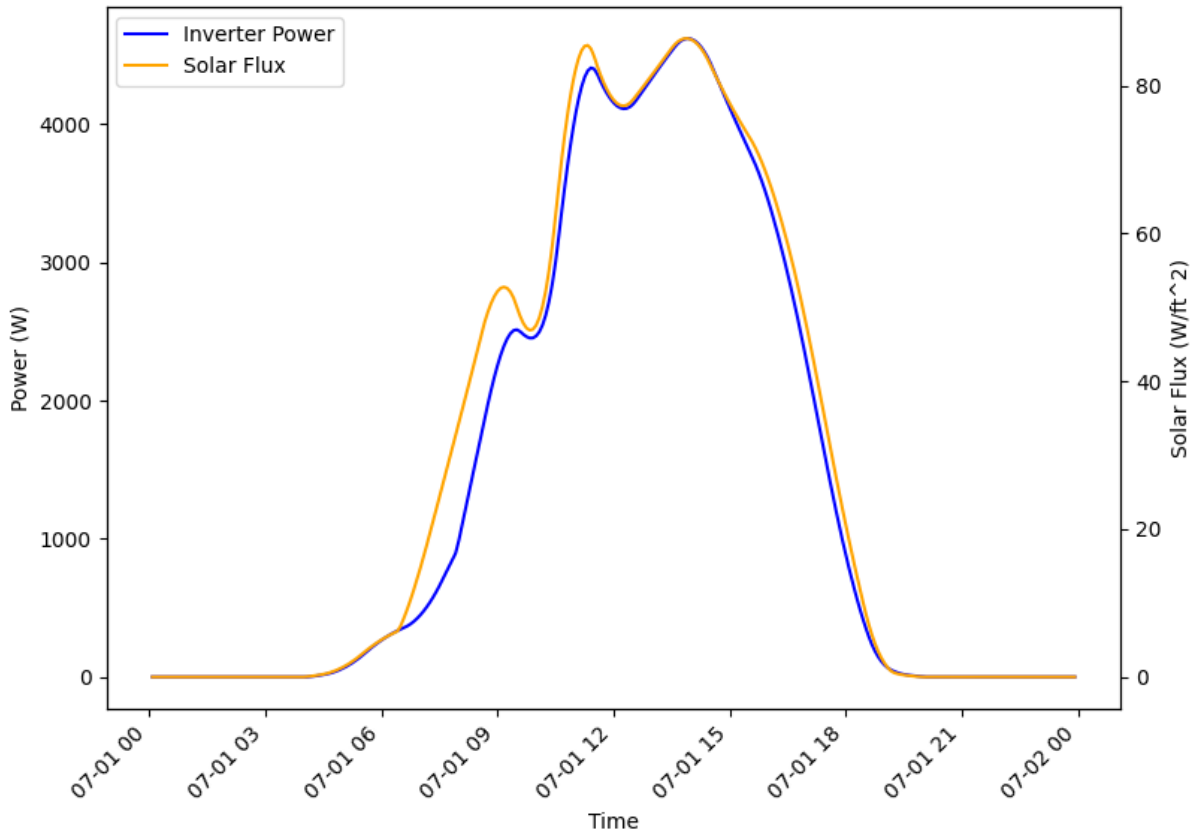


Fig. 3.7: Comparison of solar flux and rooftop solar inverter output from TE30 example.

- use **ls -al** or **cat** or **tail** on log files or csv filesto show progress of a case solution
- **./kill5570.sh** will terminate all processes connected to port 5570; if you have to do this, make sure **lsof -i :5570** shows nothing before attempting another case
- it is recommended that you append **&** to any python plot commands, so they run in the background.

3.2 TESP Example Analysis

3.2.1 SGIP1 Analysis Example

Problem Description and Analysis Goals

The Smart Grid Interoperability Panel (SGIP), as a part of its work, defined a number of scenarios for use in understanding ways in which transactive energy can be applicable. The first of these (hereafter referred to as "SGIP1") is a classic use case for transactive energy [2]:

> The weather has been hot for an extended period, and it has now reached an afternoon extreme temperature peak. Electricity, bulk-generation resources have all been tapped and first-tier DER resources have already been called. The grid operator still has back-up DER resources, including curtailing large customers on interruptible contracts. The goal is to use TE designs to incentivize more DER to participate in lowering the demand on the grid.

To flesh out this example the following additions and/or assumptions were made:

- The scenario will take place in Tucson Arizona where hot weather events that stress the grid are common.
- The scenario will span two days with similar weather but the second day will include a bulk power system generator outage that will drive real-time prices high enough to incentivize participation by transactively enabled DERs.
- Only HVACs will be used to respond to transactive signals
- Roughly 50% of the HVACs on one particular feeder will participate in the transactive system. All other feeders will be modeled by a loadshape that is not price-responsive.

The goal of this analysis are as follows:

- Evaluate the effectiveness of using transactively enabled DERs for reducing peak load.
- Evaluate the value exchange for residential customers in terms of comfort lost and monetary compensation.
- Evaluate the impacts of increasing rooftop solar PV penetration and energy storage on the transactive system performance.
- Demonstrate the capabilities of TESP to perform transactive system co-simulations.

The software version of this study implemented in TESP is similar to but not identical to earlier versions used to produce results found in [22] and [13]. As such the specific values presented here may differ from those seen in the aforementioned publications.

Valuation Model

Use Case Diagram

A Use Case Diagram is helpful in providing a broad overview of the activities taking place in the system being modeled. It shows the external actors and the specific use cases in which each participates as well as any sequencing between specific use cases.

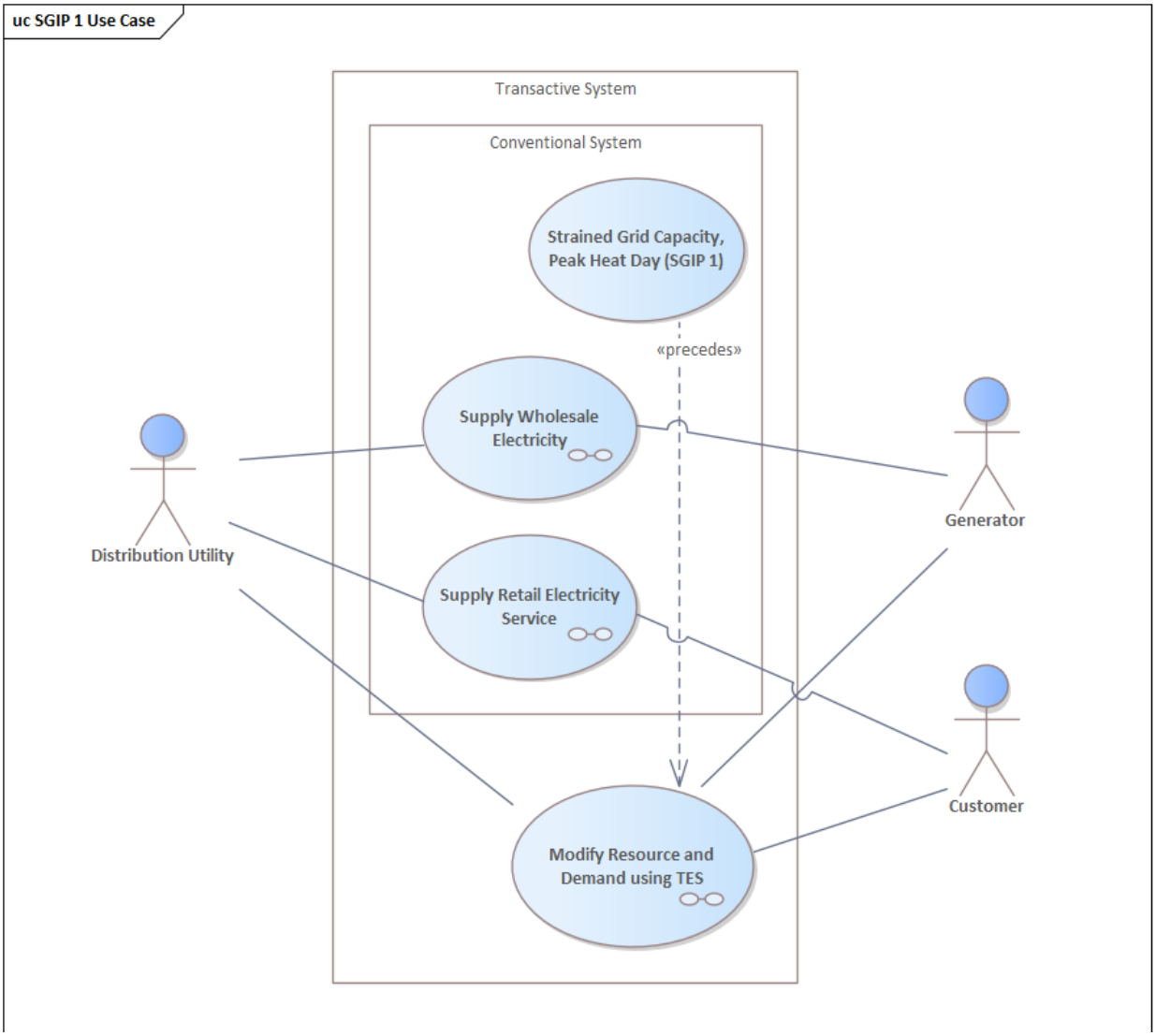


Fig. 3.8: Definition of the use cases being modeled in the system under analysis.

Value Flow Diagram

Value flows define the exchanges between actors in the system. For transactive systems, these value exchanges are essential in defining and enabling the transactive system to operate effectively. These value exchanges are often used when defining the key valuation metrics used to evaluate the performance of the system. The diagrams below show the key value exchanges modeled in this system.

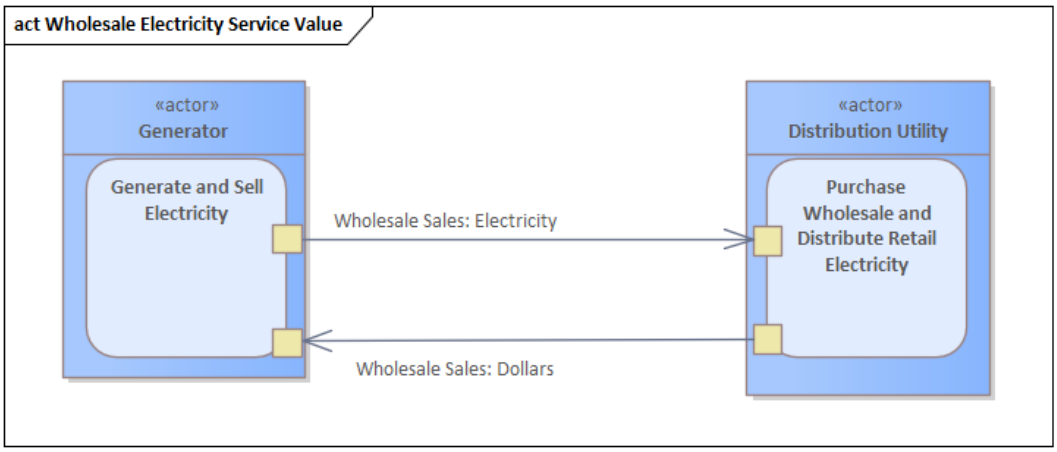


Fig. 3.9: Value exchanges modeled in the wholesale market

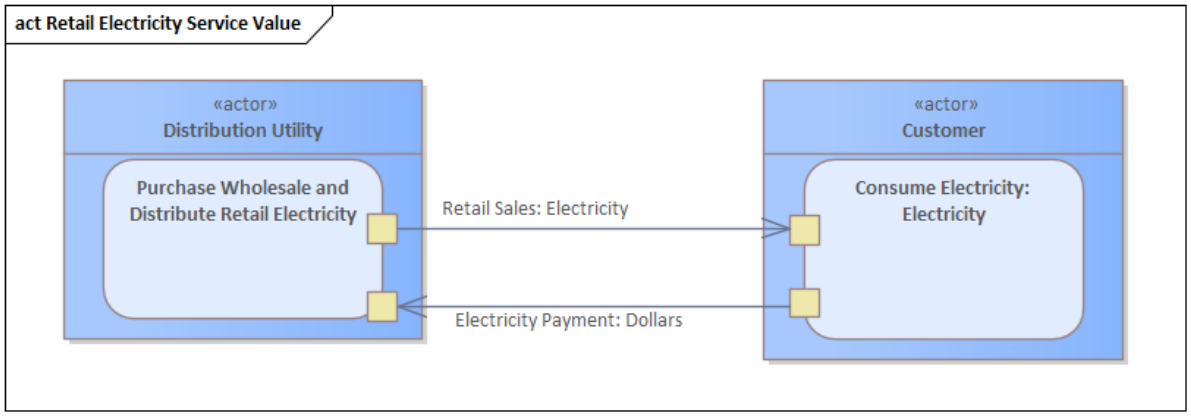
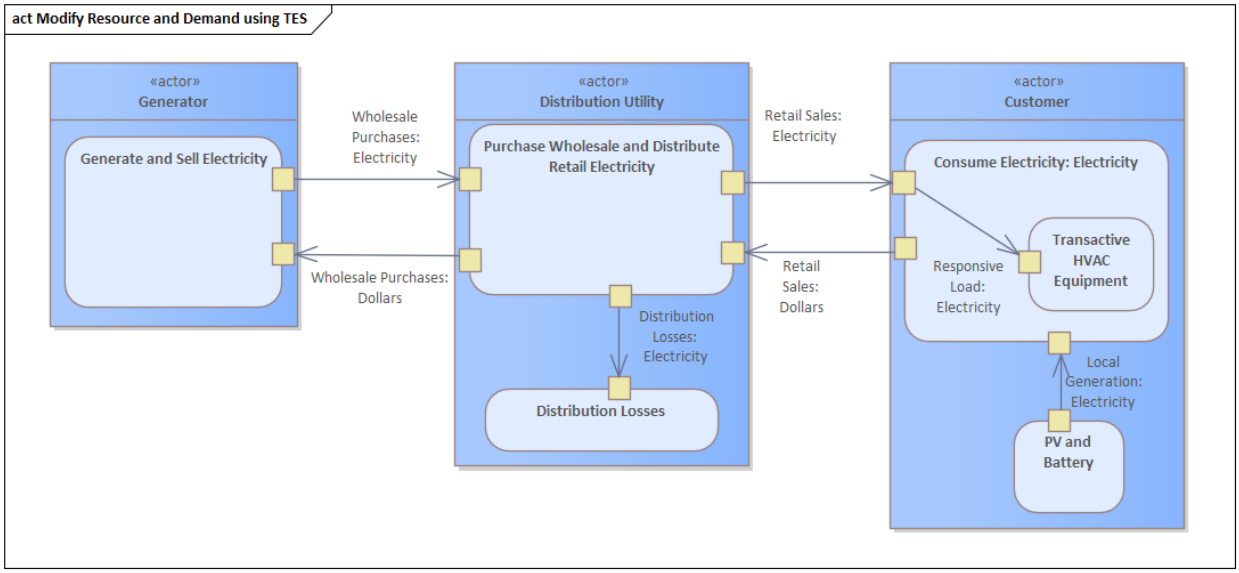


Fig. 3.10: Value exchanges modeled in the retail market

Metrics Identification

To guide the development of the analysis, it is important that key metrics are identified in the value model. The diagram below shows the specific metrics identified as sub-elements of the Accounting Table object. Though this diagram does not define the means by which these metrics are calculated, it does define the need for such a defintion, leading to a data requirement from the analysis process.

Value exchanges modeled during the transactive system operation



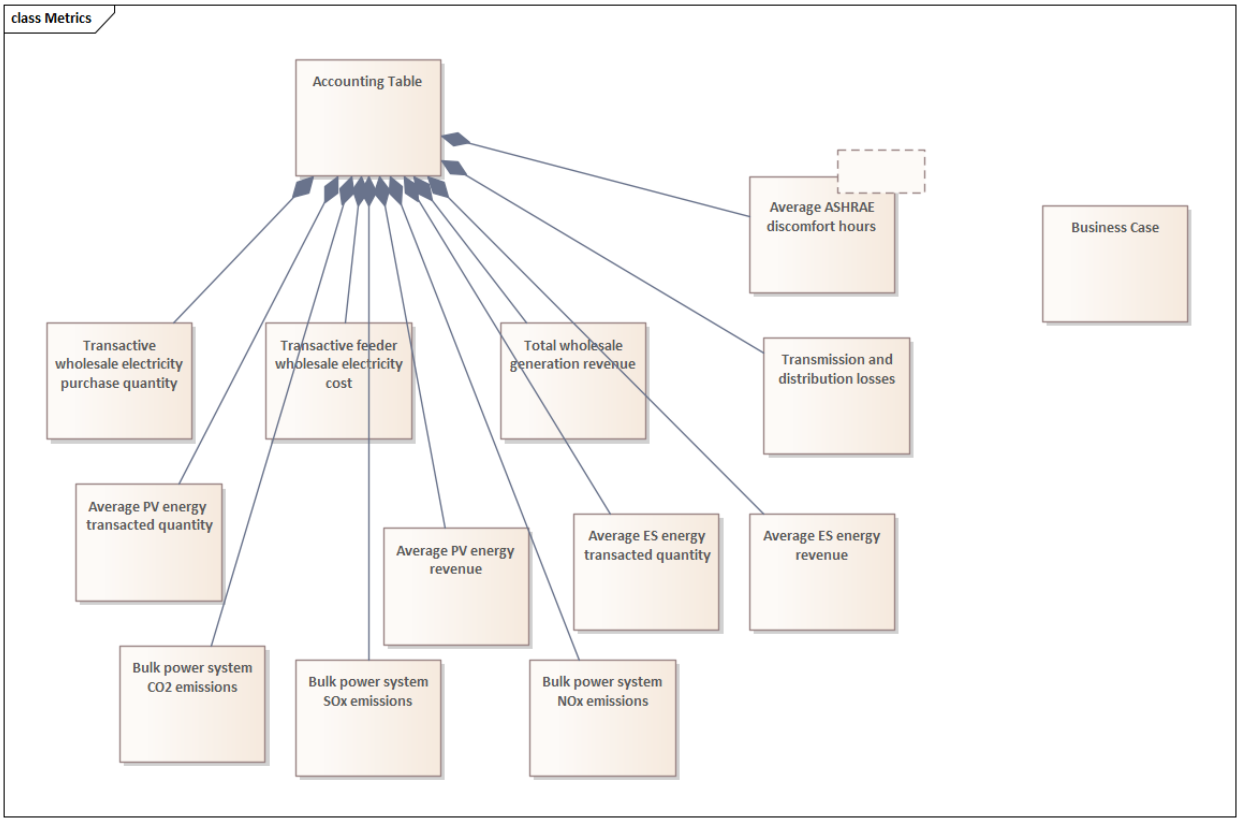


Fig. 3.11: Identification of the specific metrics to be included in the Accounting Table.

Transactive Mechanism Flowchart (Sequence Diagram)

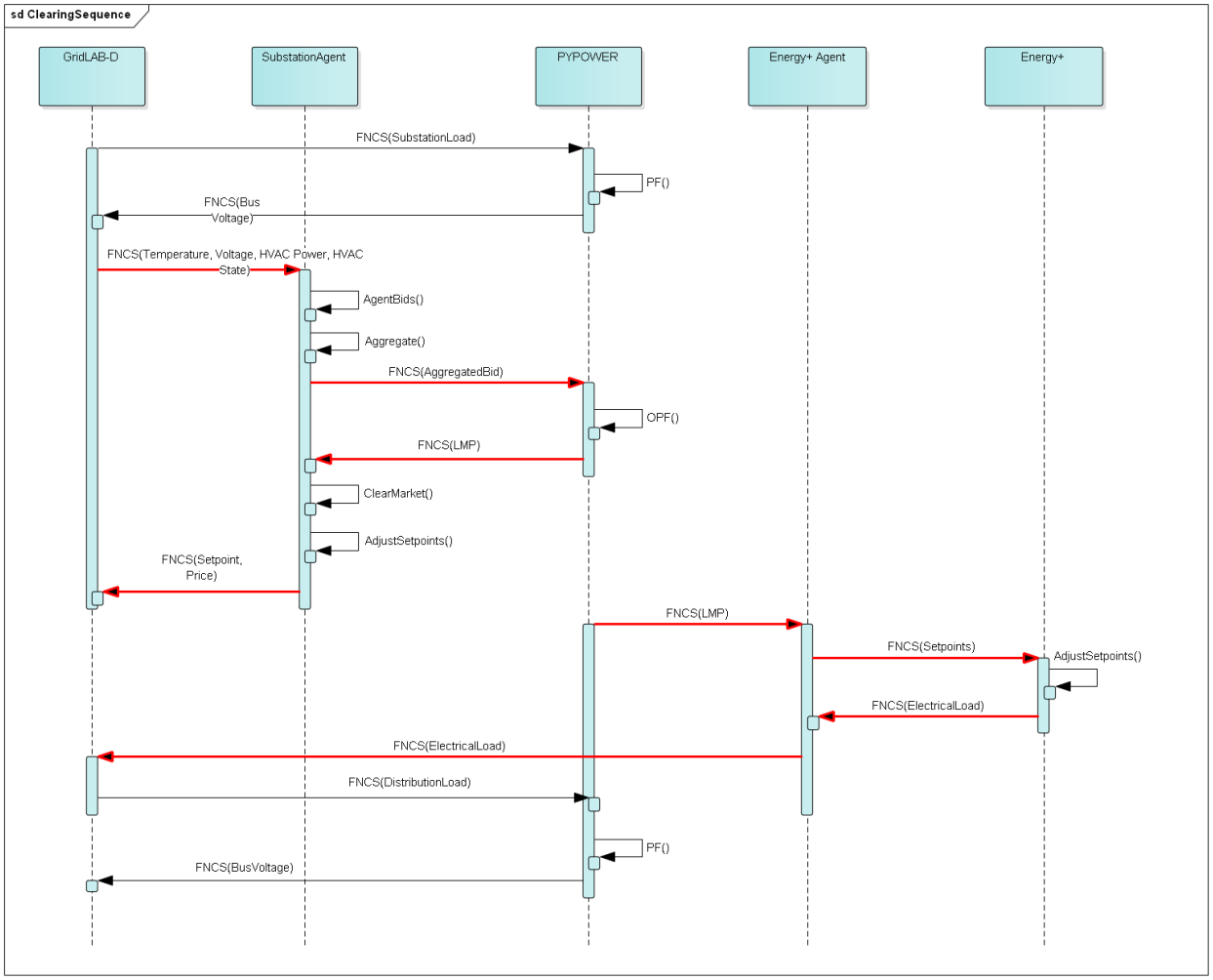


Fig. 3.12: Transactive mechanism sequence diagram showing the data exchange between participants

Key Performance Metrics Definitions

Some (but not all) of the key performance metrics used in this analysis are as follows. A list of the final metrics collected can be found in tabular form in Appendix C of [22].

Key Performance Variable Definitions:

$$
\begin{aligned}
U_i &= \text{utility functions of the individual loads} \\
C_i &= \text{utility functions of the individual generators} \\
p_i^L &= \text{power consumption of the individual loads} \\
p_j^G &= \text{power generation of the individual generators} \\
N_L &= \text{total number of loads} \\
N_G &= \text{total number of loads} \\
N_c &= \text{total number of customers} \\
t &= \text{simulated time} \\
t_{\text{day}} &= \text{last simulated time for each day} \\
P_{\text{sub}} &= \text{real power as measured at the transactive feeder's substation} \\
LMP_{\text{sub}} &= \text{price of energy at the transactive feeder's substation}
\end{aligned}
$$

Social Welfare:

$$SW = \sum_{i=1}^{N_L} U_i(p_i^L) - \sum_{j=1}^{N_G} C_j(p_j^G)$$

Electrical energy per day

$$EE_{day} = \sum_{t=0}^{t_{day}} P_{sub}$$

Electrical energy per day per customer:

$$EE_{cust \cdot day} = EE_{day}/N_c$$

Electrical energy fee per day:

$$EF_{day} = \sum_{t=0}^{t_{day}} LMP_{sub}$$

Electrical energy per day per customer:

$$EF_{cust \cdot day} = EF_{day}/N_c$$

Accounting Table Metrics Definitions

From the value model, it is possible to define metrics that will reveal the value-based outcomes of the individual participants in the transactive system. These metrics often have a financial dimension but not always. The following equations were used to produce the metrics calculated for the Accounting Table. These equations use the following definitions:

Accounting Table Variable Definitions:

Δt = time step
n_{obs} = number of daily observations
n_{days} = number of days
E_{purchase} = wholesale energy purchased at substation test feeder per day, [MWh/d]
P_{sub} = power consumed at test feeder substation, [W]
$P_{\text{generation}}$ = output power of individual generator, [MW]
aF = amp factor
E_{cost} = wholesale energy purchase cost per day, [$/d]
LMP_{purchase} = wholesale purchase price, [$/kWh]
LMP_{sell} = wholesale revenue price, [$/kWh]
$R_{\text{generation}}$ = wholesale generation revenue per day, [$/d]
$E_{\text{generation}}$ = wholesale energy generated per day, [MWh/d]
L = losses at substation, [W]
TnD = transmission and distribution losses, [% of MWh generated]
P_{PV} = PV power (positive only), [kW]
P_{ES} = ES power (positive and negative), [KW]
Y = retail clearing price, [$/kWh]
E_{PV} = average PV energy transacted, [kWh/d]
R_{PV} = average PV energy revenue, [$/d]
E_{ES} = average ES energy transacted, [kWh/d]
R_{ES} = average ES energy revenue, [$/d]
g = number of generator types in system which emit GHG out of coal, combined cycle, and single cycle
$\mathbf{R}$ = $3 \times g$ matrix of emission rates for CO_2, SO_x, and NO_x by generator type for coal, combined cycle, and single cycle
$\mathbf{G}$ = $g \times (n_{\text{obs}} \cdot n_{\text{days}})$ matrix of MWh output by generator type for coal, combined cycle, and single cycle for each interval ov
$\mathbf{K}$ = 1×3 matrix of emission conversion from lb to MT (CO_2) and kg (SO_x, NO_x)
$\mathbf{E}$ = 3×1 matrix of total emissions by GHG type (CO_2, SO_x, NO_x) over study period

Wholesale electricity purchases for test feeder (MWh/d):

$$E_{\text{purchase}} = \Delta t \cdot \frac{aF}{1 \times 10^6} \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} P_{\text{sub},i,j}$$

Wholesale electricity purchase cost for test feeder ($/day)

$$E_{\text{cost}} = \Delta t \cdot \frac{aF}{1 \times 10^3} \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} P_{\text{sub},i,j} \cdot LMP_{\text{purchase},i,j}$$

Total wholesale generation revenue ($/day)

$$R_{\text{generation}} = \Delta t \cdot 1 \times 10^3 \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} P_{\text{generation},i,j} \cdot LMP_{\text{sell},i,j}$$

Transmission and Distribution Losses (% of MWh generated):

$$TnD = \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} \frac{L_{i,j}}{P_{\text{sub},i,j}}$$

Average PV energy transacted (kWh/day):

$$E_{\text{PV}} = \frac{\Delta t}{n_{\text{obs}}} \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} P_{\text{PV},i,j}$$

Average PV energy revenue ($/day):

$$R_{\text{PV}} = \frac{\Delta t}{n_{\text{obs}}} \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} Y_{i,j} \cdot P_{\text{PV},i,j}$$

Average ES energy transacted (kWh/day):

$$E_{\text{ES}} = \frac{\Delta t}{n_{\text{obs}}} \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} P_{\text{ES},i,j}$$

Average ES energy net revenue:

$$R_{\text{ES}} = \frac{\Delta t}{n_{\text{obs}}} \cdot \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} Y_{i,j} \cdot P_{\text{ES},i,j}$$

Emissions:

Table 3.2: Emissions Concentrations by Technology Type

	CO2	SOX	NOX
coal	2074.2013	1.009	0.6054
combined cycle	898.0036	0.00767	0.057525
single cycle	1331.1996	0.01137	0.085275

Table 3.3: Conversion from lb to MT (CO2) and kg (SOx, NOx)

	K
CO2	0.000453592
SOx	0.453592
NOx	0.453592

Total CO2, SOx, NOx emissions (MT/day, kg/day, kg/day):

$$\mathbf{E} = \sum_{i=1}^{n_{\text{days}}} \sum_{j=1}^{n_{\text{obs}}} \sum_{k=1}^{g} \mathbf{G}_{i,j,k} \times \mathbf{R}_k \times \mathbf{K}$$

Analysis Design Model

The analysis design model is a description of the planned analysis process showing how all the various analysis steps lead towards the computation of the key performance metrics. The data requirements of the valuation and validation metrics drive the definition of the various analysis steps that must take place in order to be able to calculate these metrics.

The level of detail is in this model is somewhat subjective and up to those leading the analysis. There must be sufficient detail to avoid the biggest surprises when planning the execution of the analysis but a highly detailed plan is likely to be more effort than it is worth. The analysis design model supports varying levels of fidelity by allowing any individual activity block to be defined in further detail through the definition of subactivities

Top Level

The top level analysis diagram (shown in Fig. 3.13) is the least detailed model and shows the analysis process at the coarsest level. On the left-hand side of the diagram is the source data (which includes assumptions) and is the only analysis activity with no inputs. The analysis activity blocks in the middle of the diagram show the creation of various outputs from previously created inputs with the terminal activities being the presentation of the final data in the form of tables, graphs, and charts.

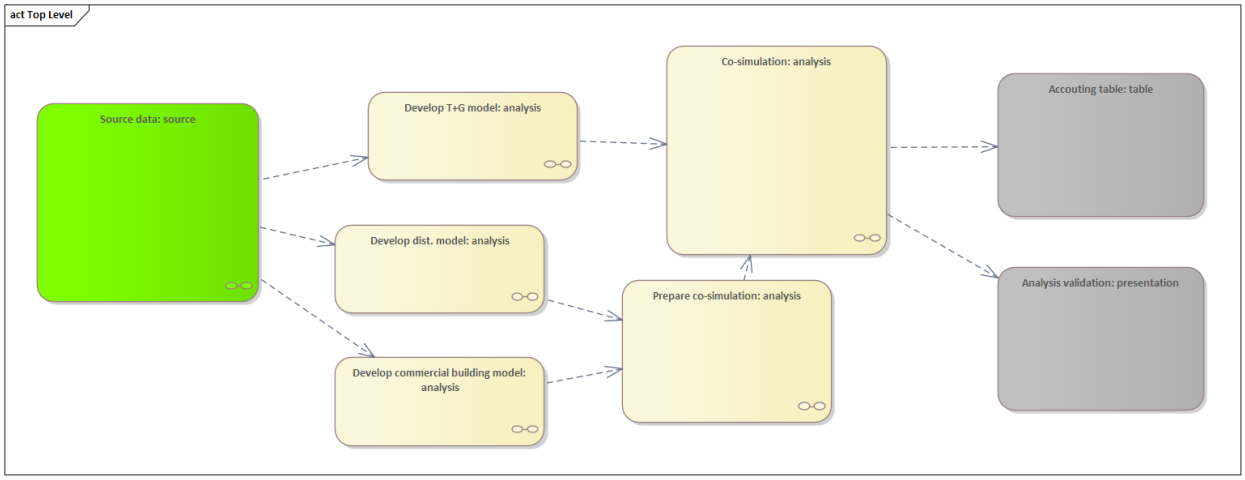


Fig. 3.13: Top level view of the analysis design model

Source Data

The green source data block in the top level diagram (see Fig. 3.13) is defined in further detail in a sub-diagram shown in Fig. 3.14. Many of these items are more than single values and are more complex data structures themselves.

Develop Transmission and Generation Model

The "Develop T+G model" activity block in the top level diagram (see Fig. 3.13) is defined in further detail in a sub-diagram shown in Fig. 3.15. The diagram shows that both generation and transmission network information is used to create a PYPOWER model.

Develop Distribution Model

The "Develop dist. model" activity block in the top level diagram (see Fig. 3.13) is defined in further detail in a sub-diagram shown in Fig. 3.16. The distribution model uses assumptions and information from the Residential Energy Consumer Survey (RECS) to define the properties of the modeled houses as well as the inclusion of rooftop solar PV and the participation in the transactive system. These inputs are used to generate a GridLAB-D model.

Fig. 3.14: Detailed view of the data sources necessary to the SGIP1 analysis.

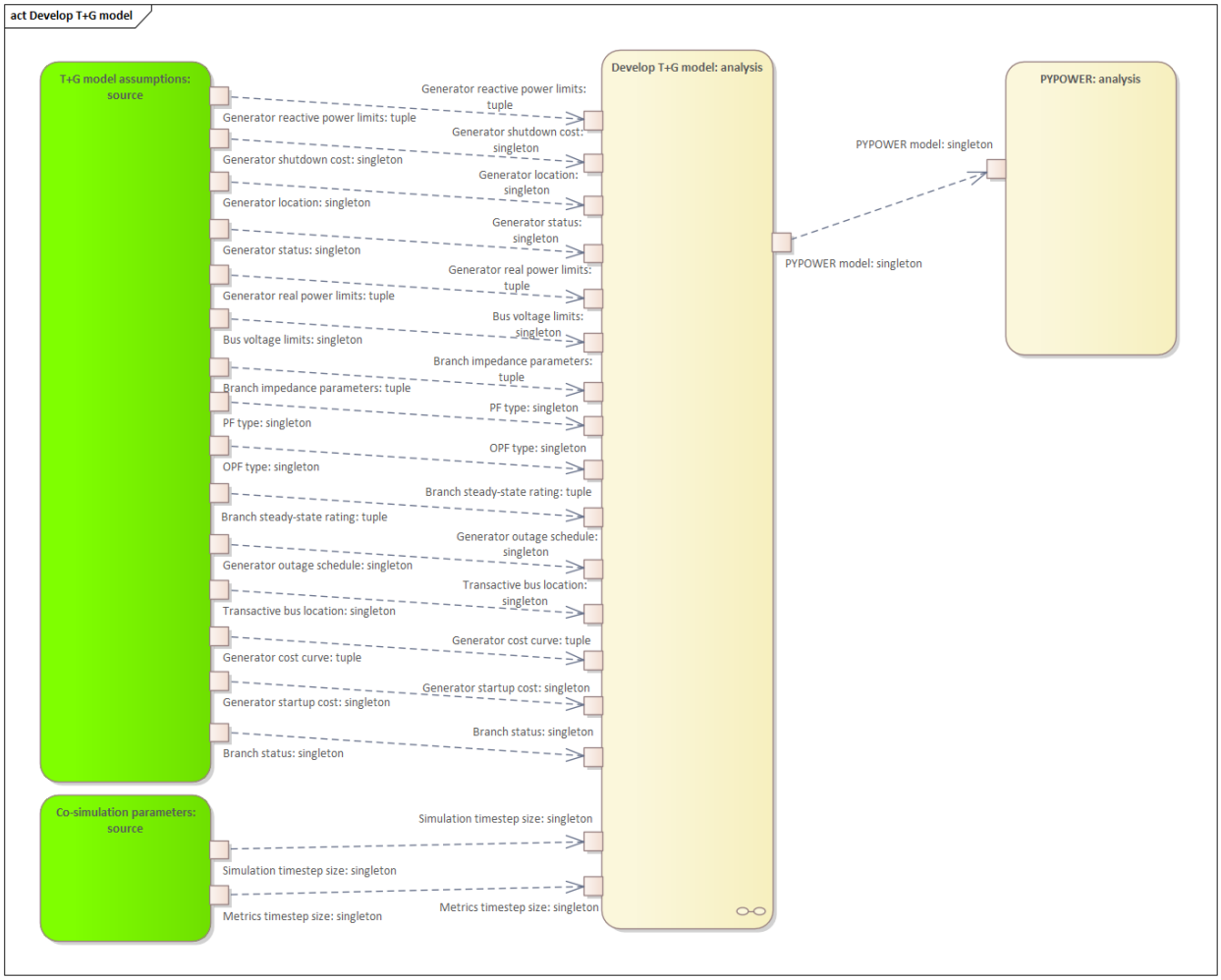


Fig. 3.15: Detailed model of the development process of the transmission and generation system model.

Fig. 3.16: Detailed model of the development process of the distribution system model.

Develop Commercial Building Model

The "Develop commercial building model" activity block in the top level diagram (see Fig. 3.13) is defined in further detail in a sub-diagram shown in Fig. 3.17. The commercial building model is a predefined Energy+ model paired with a particular TMY3 weather file (converted to EPW for use in Energy+).

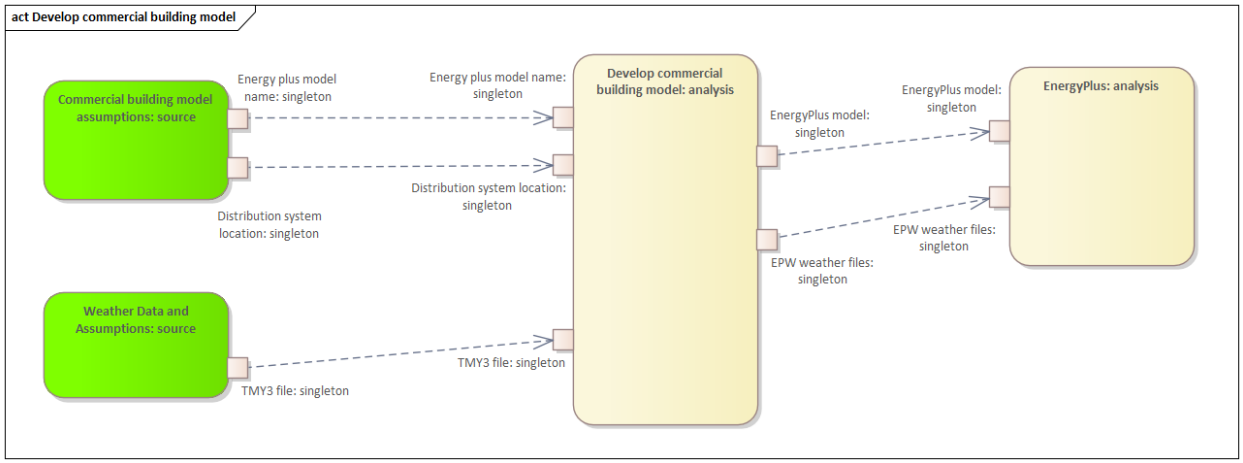


Fig. 3.17: Detailed model of the development process of the commercial building.

Prepare co-simulation

The "Prepare co-simulation" activity block in the top level diagram (see Fig. 3.13) is defined in further detail in a sub-diagram shown in Fig. 3.18. The core activity is the "Create co-sim config files" which are used by their respective simulation tools. Additionally, a special metadata file is created from the GridLAB-D model and is used by several of the metrics calculations directly.

Co-simulation

The "Co-simulation" activity block in the top level diagram (see Fig. 3.13) is defined in further detail in a sub-diagram shown in Fig. 3.19. The GridLAB-D model plays a central role as a significant portion of the modeling effort is centered around enabling loads (specifically HVACs) to participate in the transactive system. In addition to the previously shown information flows between the activities the dynamic data exchange that takes place during the co-simulation run; this is shown by the "<<flow>>" arrows.

Accounting Table

The "Accounting table" presentation block in the top level diagram (see Fig. 3.13) is defined in further detail in a series of sub-diagrams shown below. Each line of the accounting table shown in Fig. 3.11 is represented by a gray "presentation" block, showing the required inputs to produce that metric.

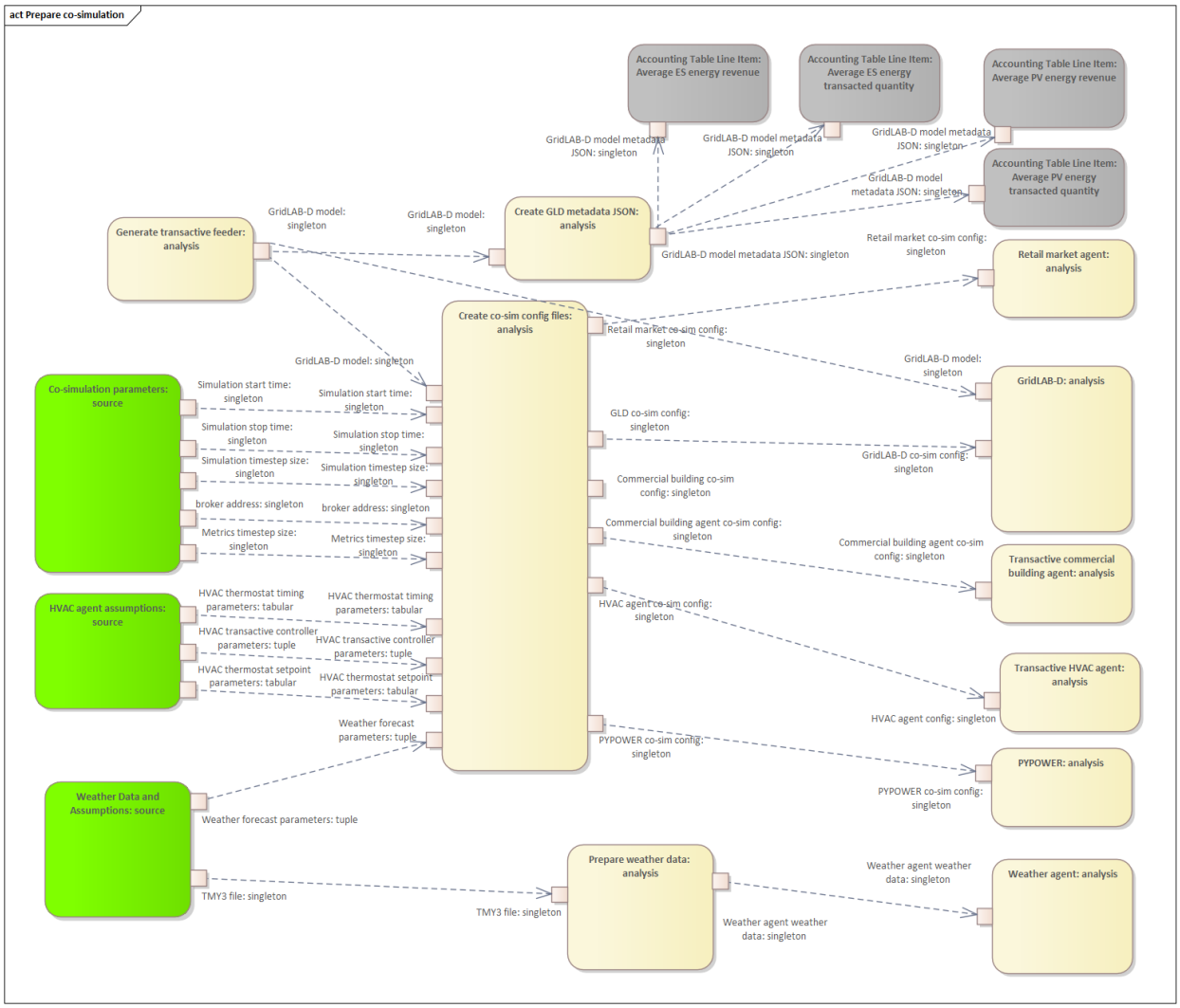


Fig. 3.18: Detailed model of the co-simulation configuration file creation.

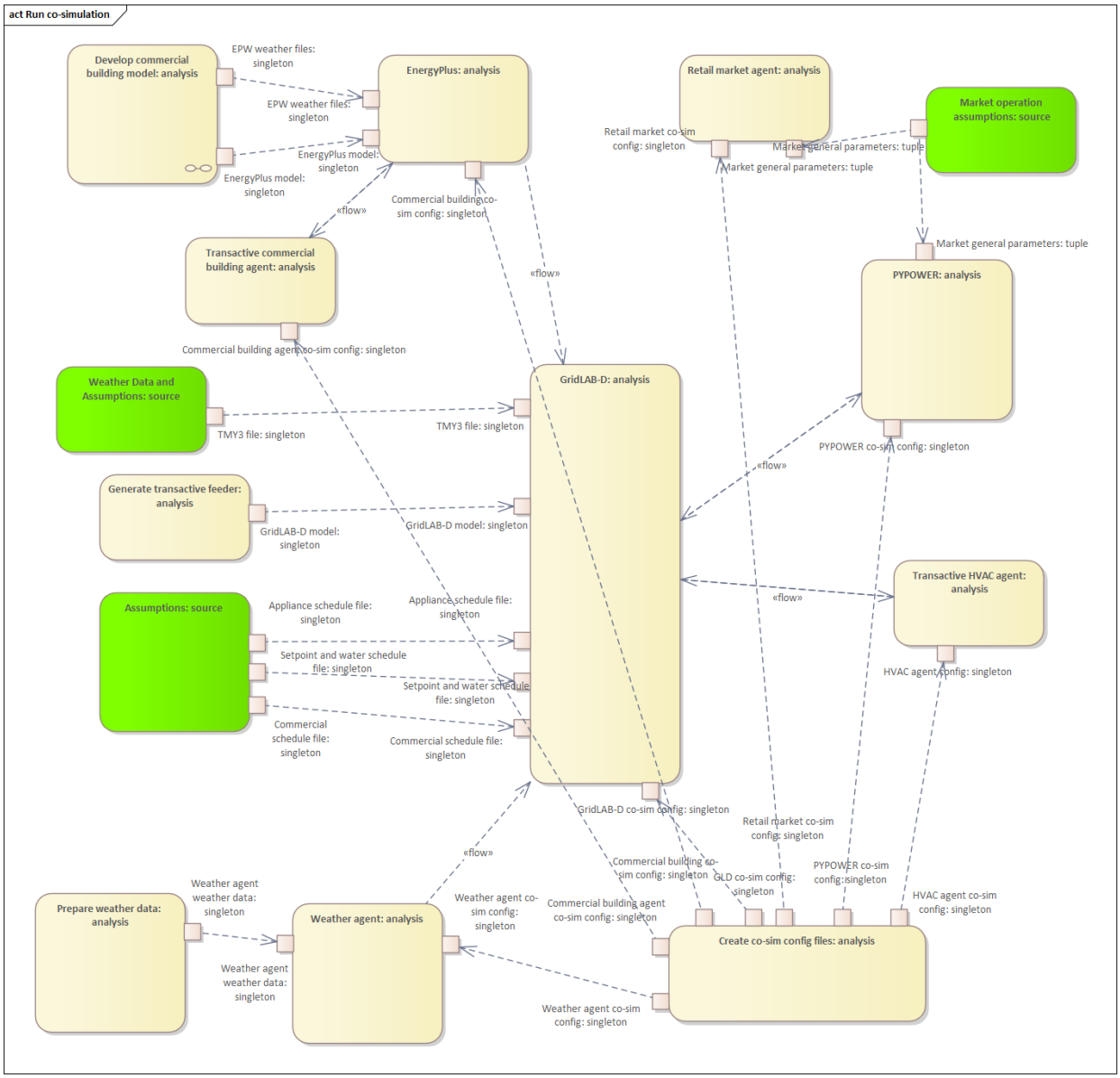


Fig. 3.19: Detailed model of the co-simulation process showing the dynamic data exchanges with “<<flow>>” arrows.

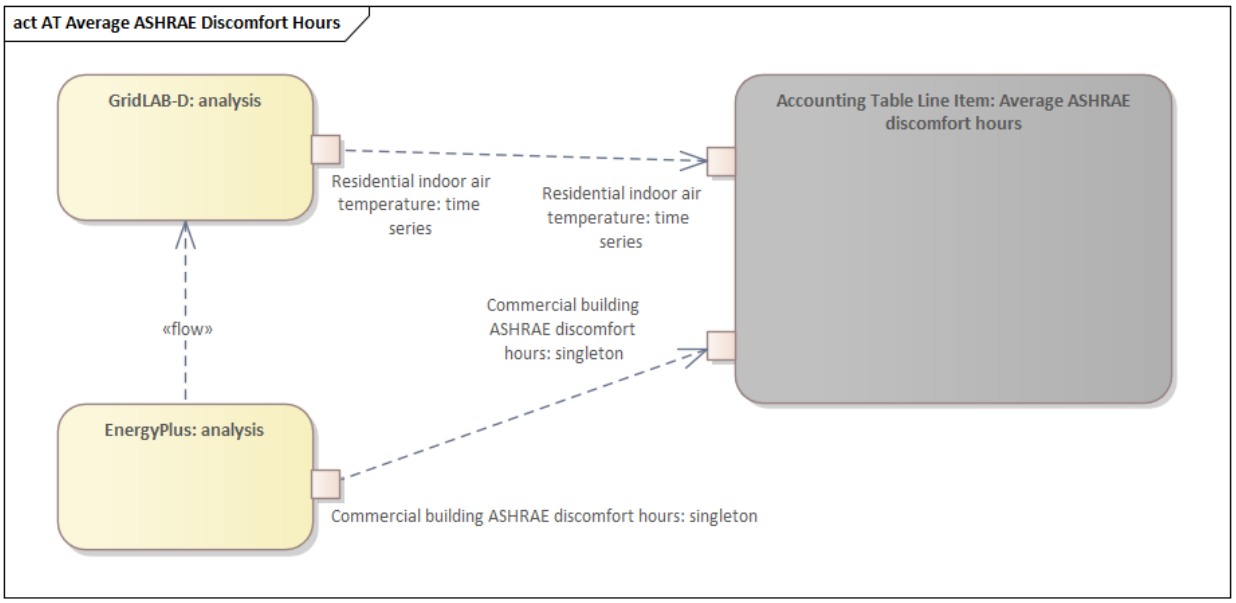


Fig. 3.20: Average ASHRAE discomfort hours metric data flow

Analysis Validation

The "Analysis validation" presentation block in the top level diagram (see Fig. 3.13) is defined in further detail in a series of sub-diagrams shown below. These are metrics similar to those in the *Accounting Table* section but they are not necessarily defined by the value exchanges and thus fall outside the value model. These metrics are identified by the analysis designer in cooperation with analysis team as a whole and are used to validate the correct execution of the analysis.

Simulated System Model

Fig. 3.29 shows the types of assets and stakeholders considered for the use cases in this version. The active market participants include a double-auction market at the substation level, the bulk transmission and generation system, a large commercial building with one-way (price-responsive only) HVAC thermostat, and single-family residences that have a two-way (fully transactive) HVAC thermostat. Transactive message flows and key attributes are indicated in **orange**.

In addition, the model includes residential rooftop solar PV and electrical energy storage resources at some of the houses, and waterheaters at many houses. These resources can be transactive, but are not in this version. The rooftop solar PV has a nameplate efficiency of 20% and inverters with 100% efficiency. inverters are set to operate at a constant power factor of 1.0. The rated power of the rooftop solar PV installations varies from house to house and ranges from roughly 2.7 kW to 4.5 kW.

The energy storage devices also have inverters with 100% efficiency and operate in an autonomous load-following mode that performs peak-shaving and valley-filling based on the total load of the customer's house to which it is attached. All energy storage devices are identical with a capacity of 13.5 kWh and a rated power of 5 kW (both charging and discharging). The batteries are modeled as lithium-ion batteries with a round-trip efficiency of 86%. Other details can be found in Table 3.4.

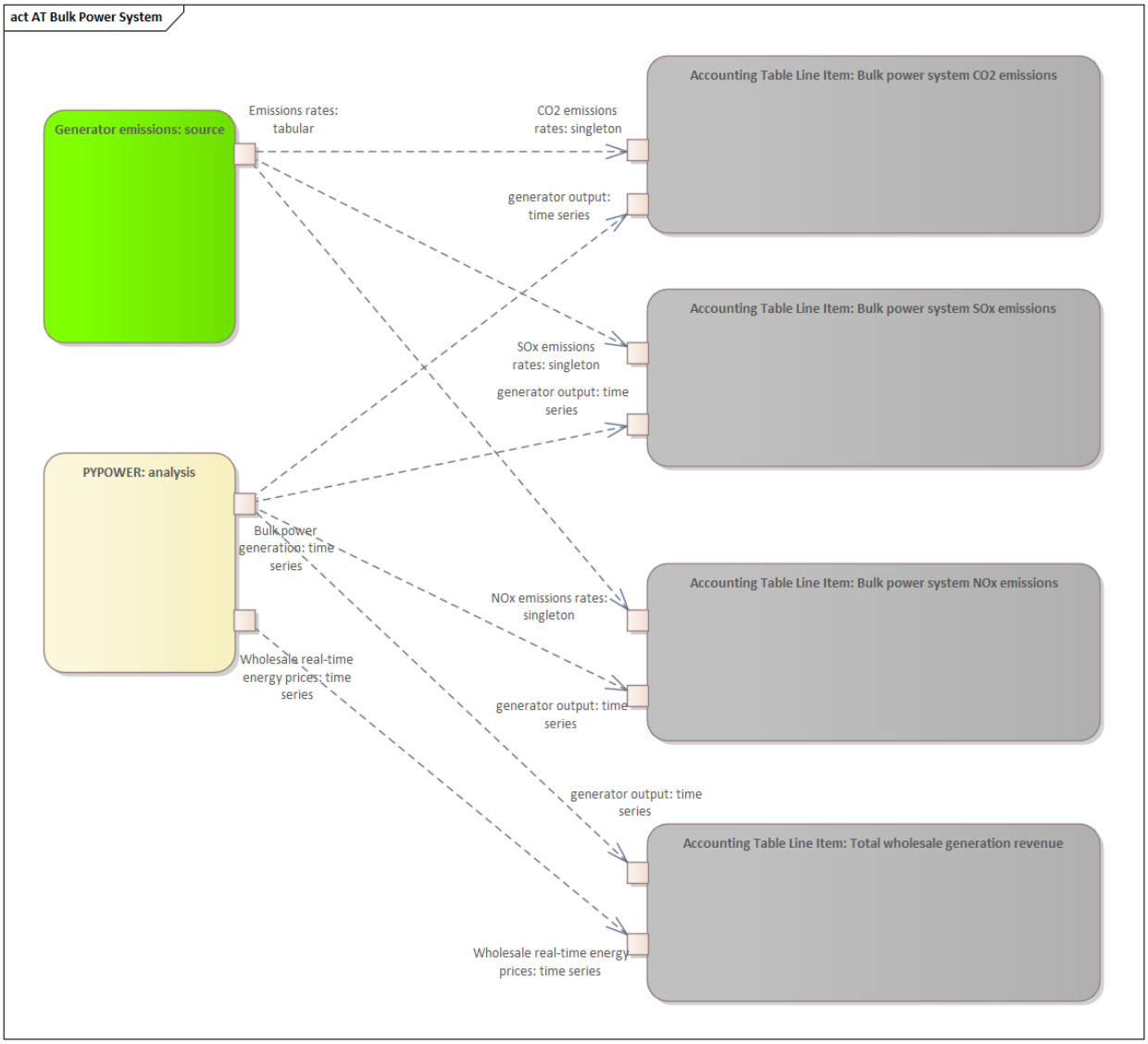


Fig. 3.21: Bulk power system (T+G) metrics data flows

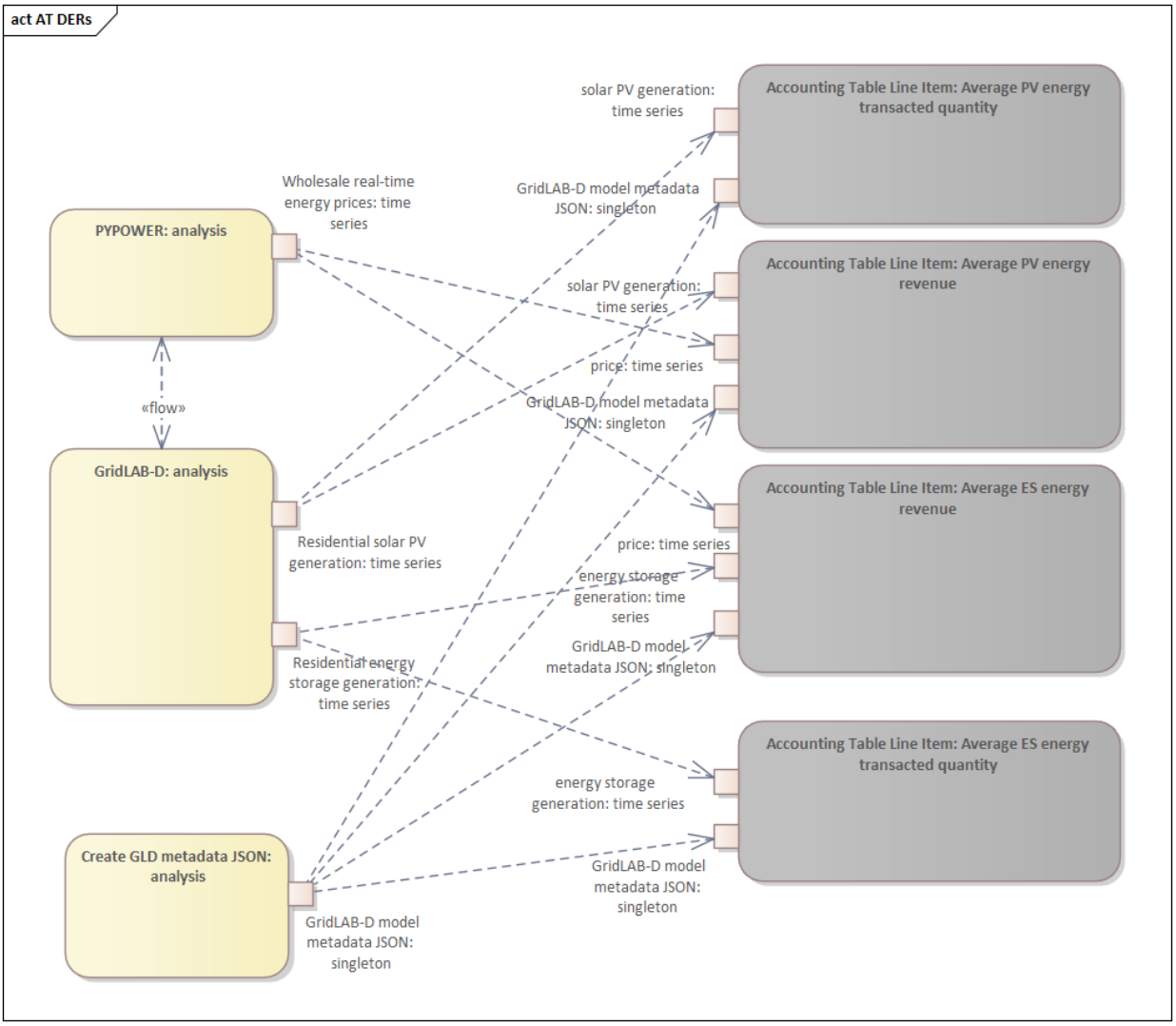


Fig. 3.22: Distributed energy resources (DERs) metrics data flows

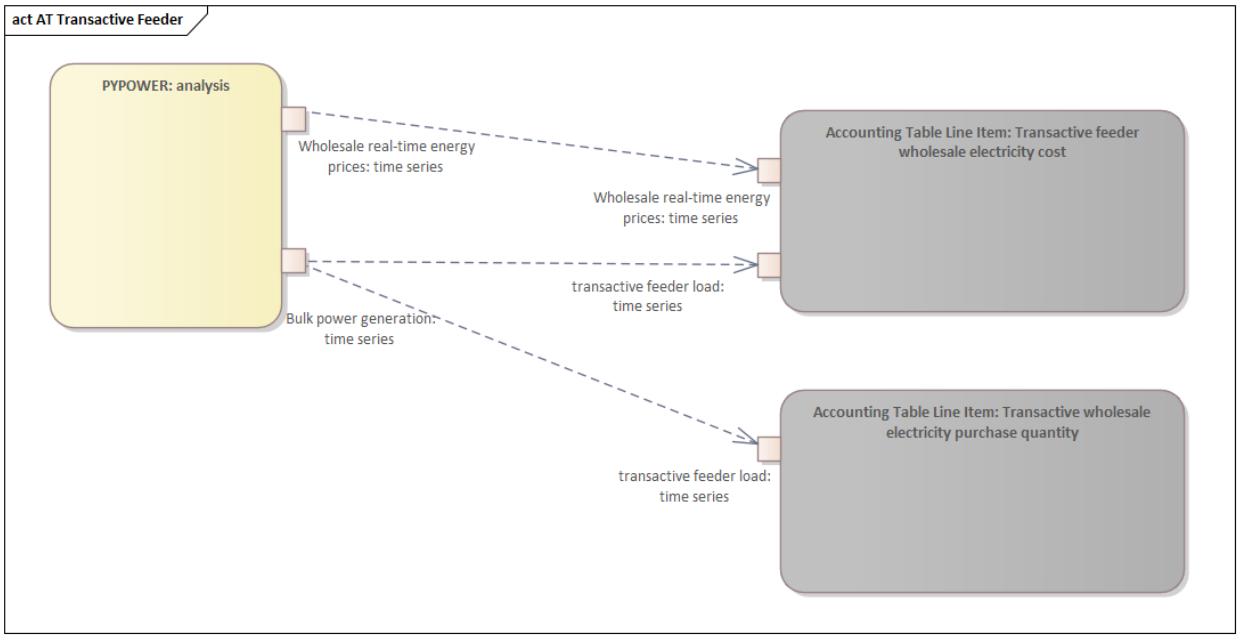


Fig. 3.23: Transactive feeder metric data flows

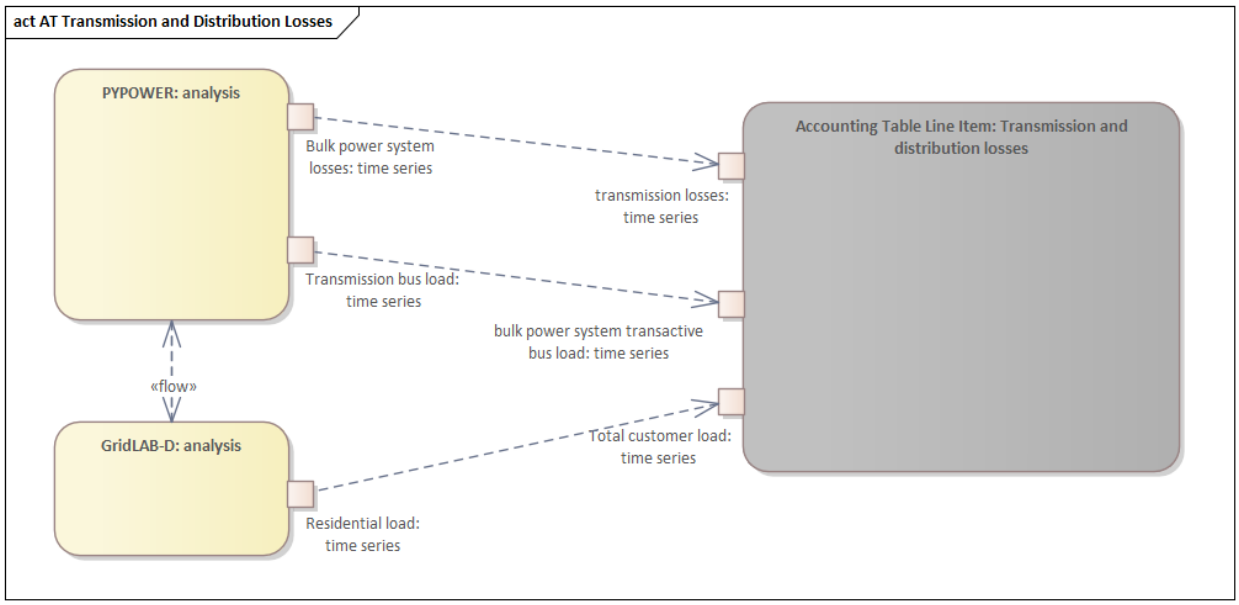


Fig. 3.24: Transmission and distribution network losses metric data flows

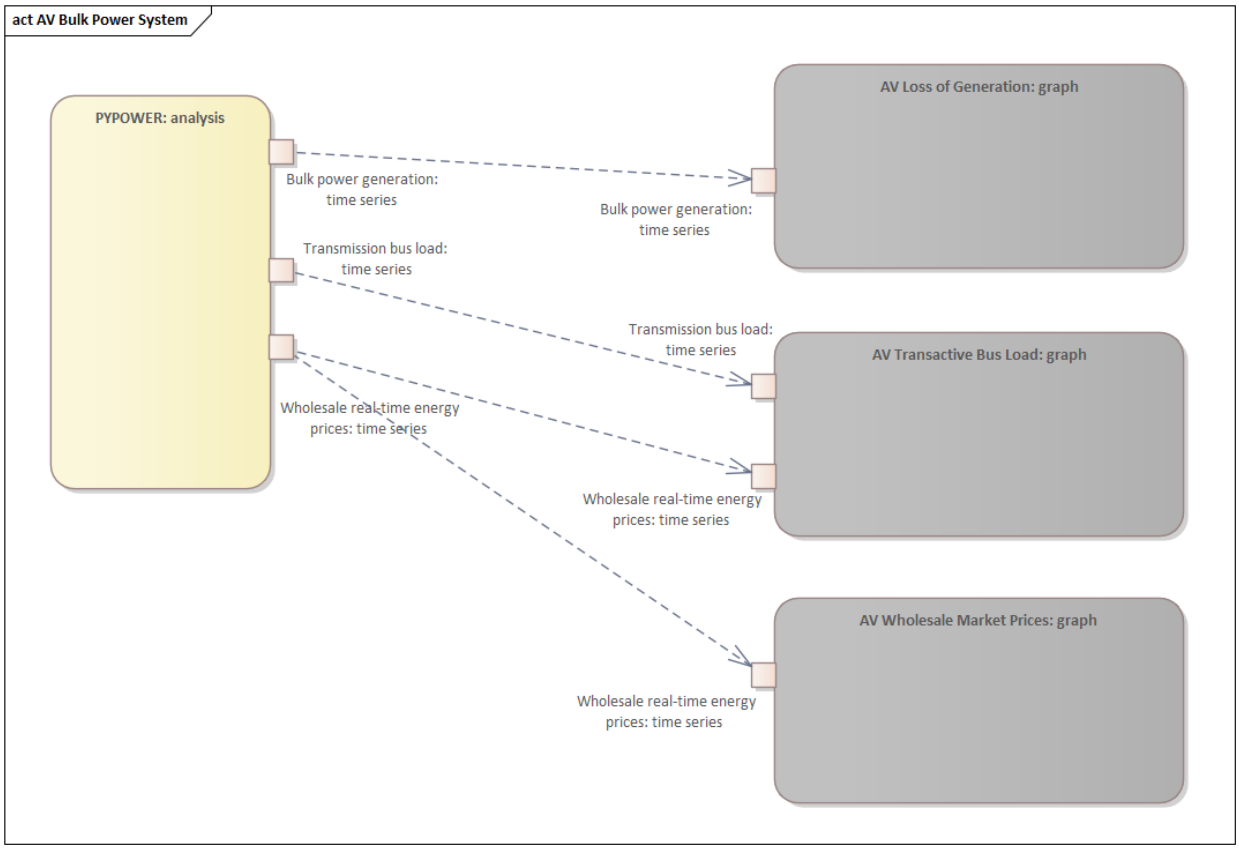


Fig. 3.25: Bulk power system metrics data flows

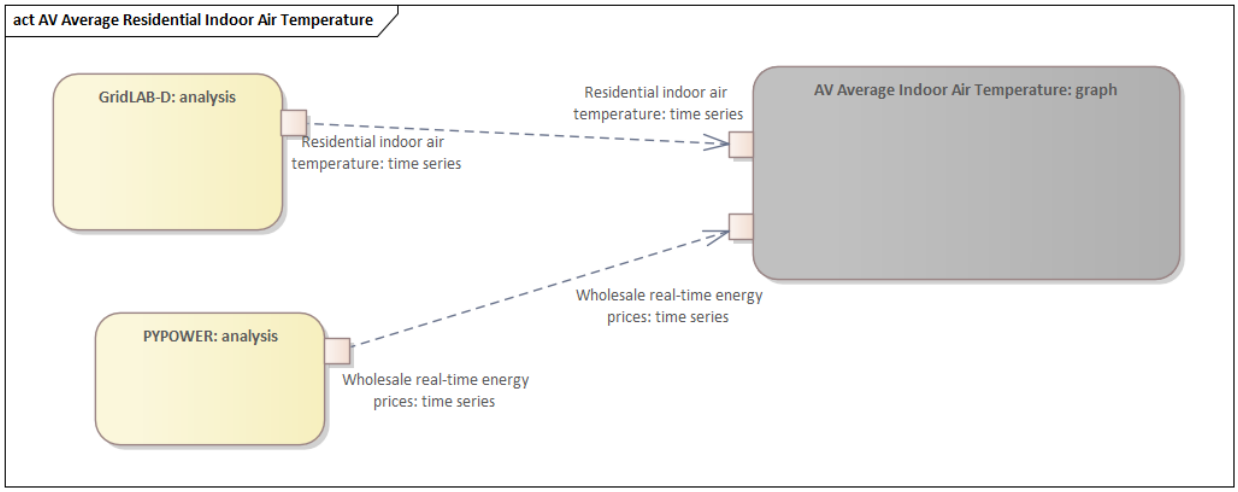


Fig. 3.26: Residential indoor air temperature metric data flows

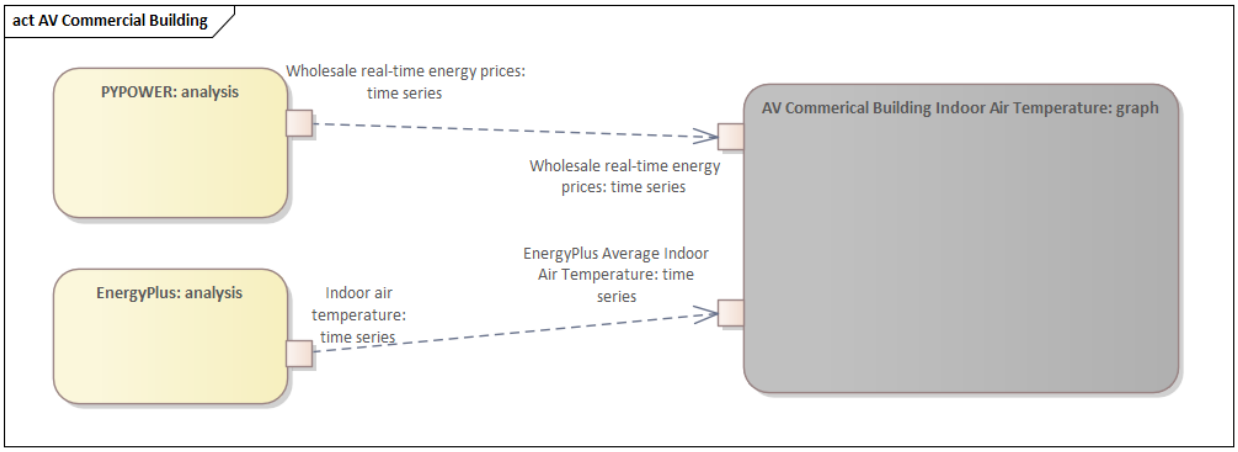


Fig. 3.27: Commercial indoor air temperature metric data flows

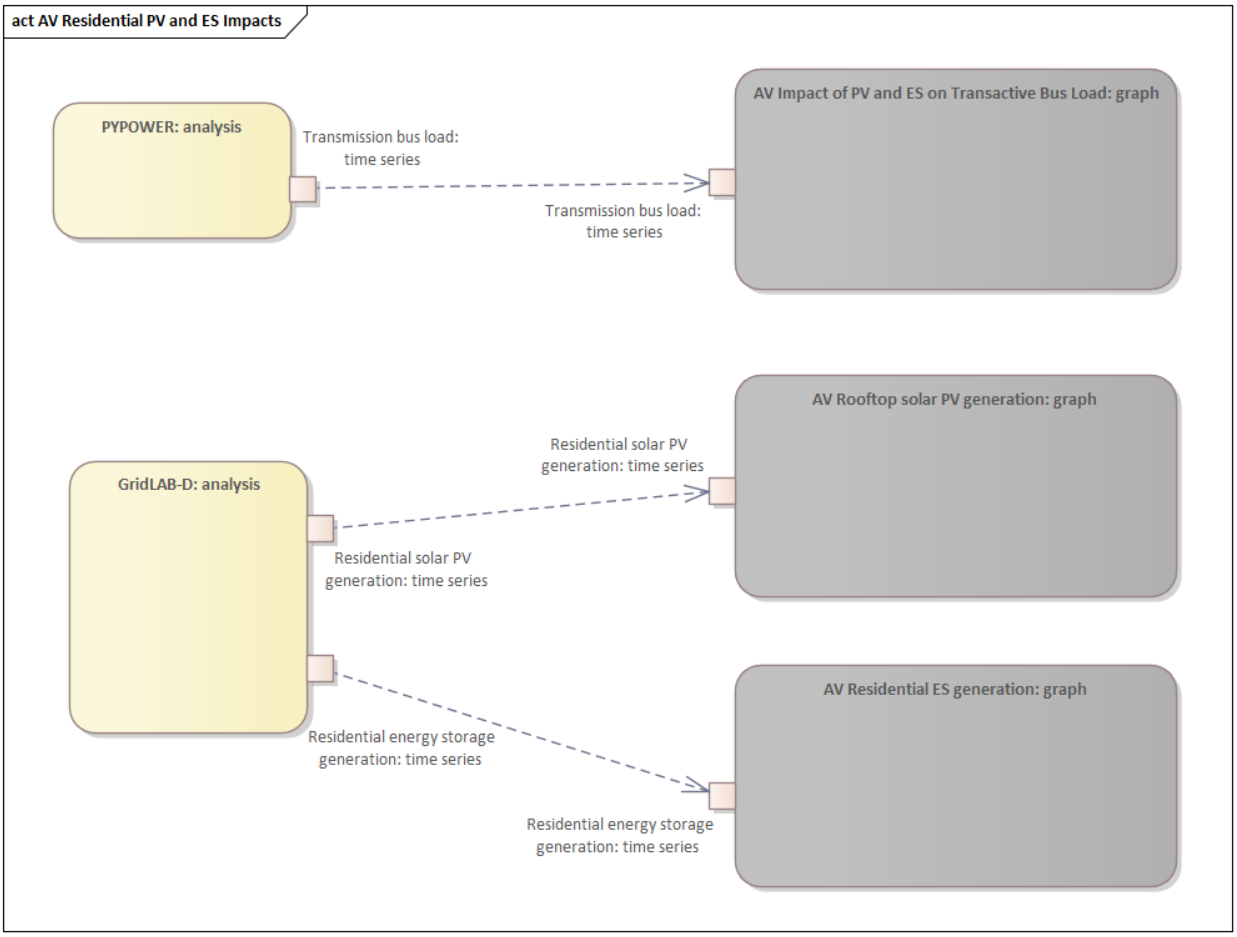


Fig. 3.28: Residential rooftop solar PV and energy storage metrics data flows

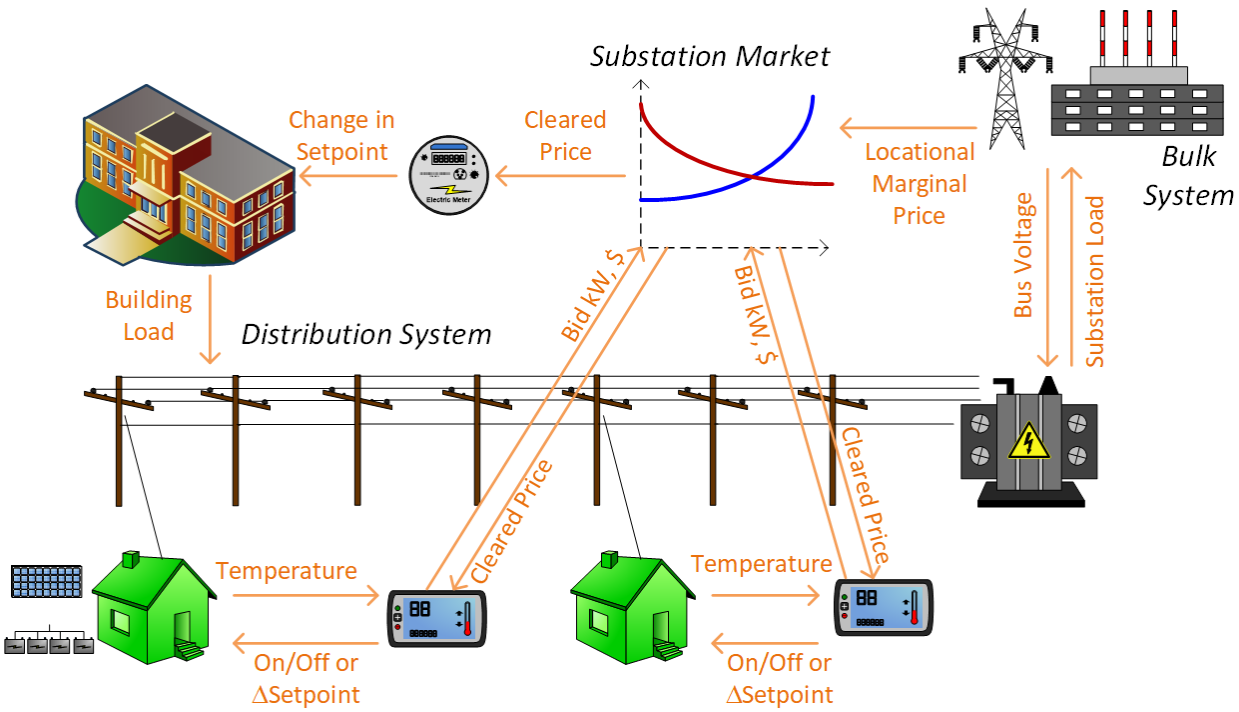


Fig. 3.29: SGIP-1 system configuration with partial PV and storage adoption

The Circuit Model

Fig. 3.30 shows the bulk system model in PYPOWER. It is a small system with three generating units and three load buses that comes with PYPOWER, to which we added a high-cost peaking unit to assure convergence of the optimal power flow in all cases. In SGIP-1 simulations, generating unit 2 was taken offline on the second day to simulate a contingency. The GridLAB-D model was connected to Bus 7, and scaled up to represent multiple feeders. In this way, prices, loads and resources on transmission and distribution systems can impact each other.

Fig. 3.31 shows the topology of a 12.47-kV feeder based on the western region of PNNL's taxonomy of typical distribution feeders [26]. We use a MATLAB feeder generator script that produces these models from a typical feeder, including random placement of houses and load appliances of different sizes appropriate to the region. The model generator can also produce small commercial buildings, but these were not used here in favor of a detailed large building modeled in EnergyPlus. The resulting feeder model included 1594 houses, 755 of which had air conditioning, and approximately 4.8 MW peak load at the substation. We used a typical weather file for Arizona, and ran the simulation for two days, beginning midnight on July 1, 2013, which was a weekday. A normal day was simulated in order for the auction market history to stabilize, and on the second day, a bulk generation outage was simulated. See the code repository for more details.

Fig. 3.32 shows the building envelope for an elementary school model that was connected to the GridLAB-D feeder model at a 480-volt, three-phase transformer secondary. The total electric load varied from 48 kW to about 115 kW, depending on the hour of day. The EnergyPlus agent program collected metrics from the building model, and adjusted the thermostat setpoints based on real-time price, which is a form of passive response.

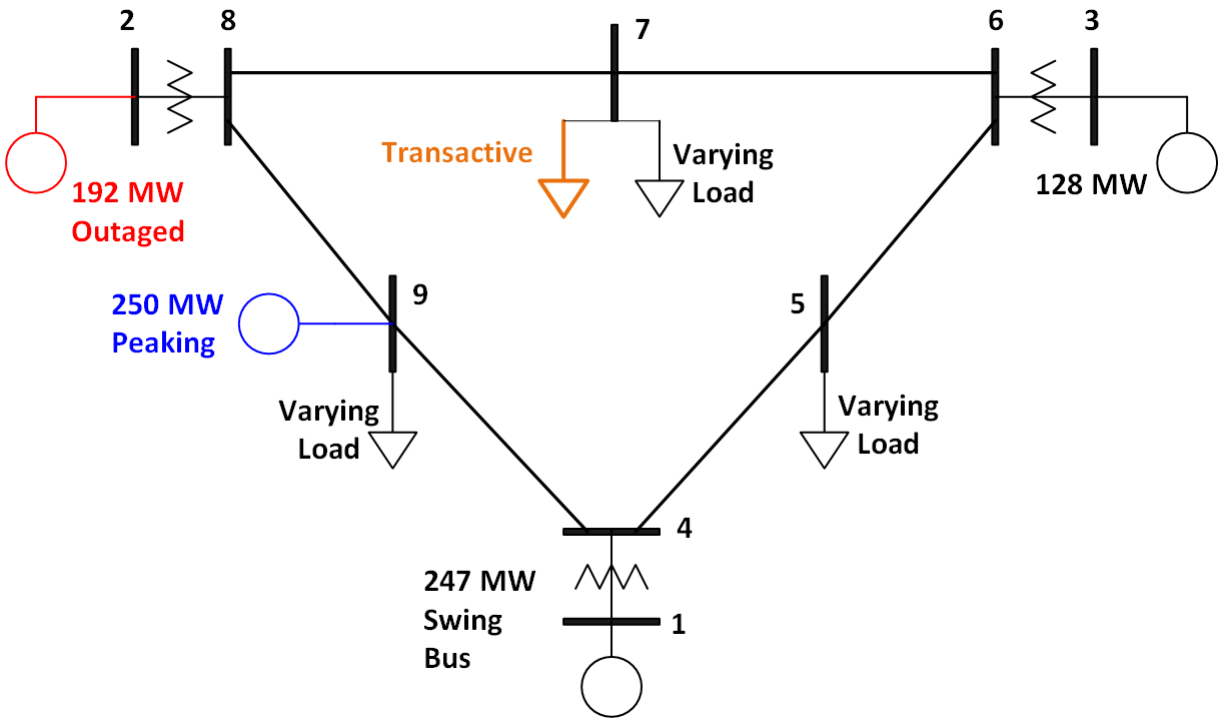


Fig. 3.30: Bulk System Model with Maximum Generator Real Power Output Capacities

The Growth Model

This version of the growth model has been implemented for yearly increases in PV adoption, storage adoption, new (greenfield) houses, and load growth in existing houses. For SGIP-1, only the PV and storage growth has actually been used. A planned near-term extension will cover automatic transformer upgrades, making use of load growth more robust and practical.

Table 3.4 summarizes the growth model used in this report for SGIP-1. In row 1, with no (significant) transactive mechanism, one HVAC controller and one auction market agent were still used to transmit PYPOWER's LMP down to the EnergyPlus model, which still responded to real-time prices. In this version, only the HVAC controllers were transactive. PV systems would operate autonomously at full output, and storage systems would operate autonomously in load-following mode.

Fig. 3.31: Distribution Feeder Model (http://emac.berkeley.edu/gridlabd/taxonomy_graphs/)

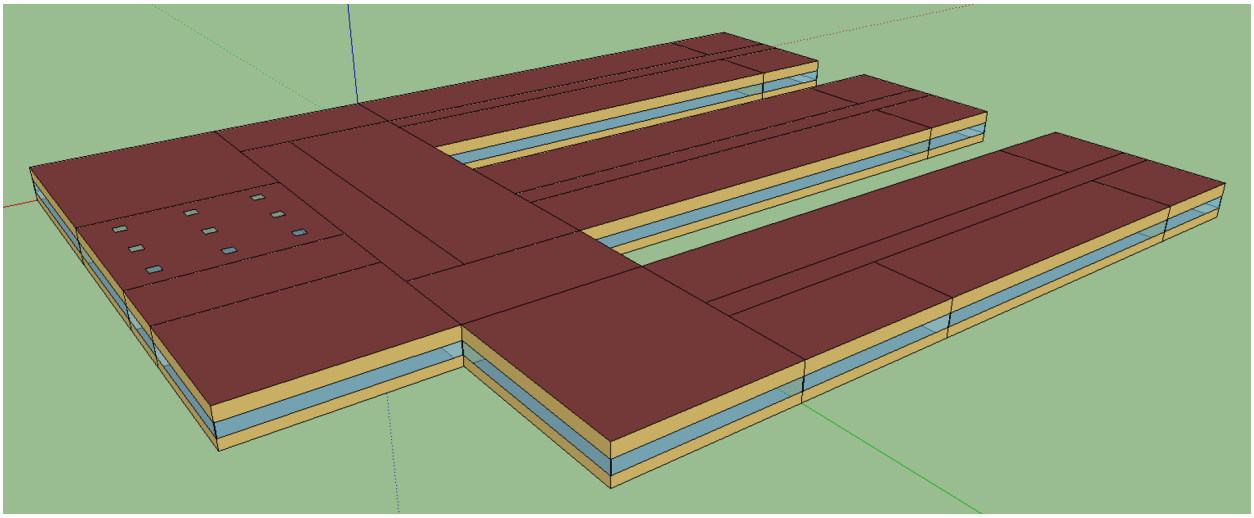


Fig. 3.32: Elementary School Model

Table 3.4: Growth Model for SGIP-1 Simulations

Case	Houses	HVAC Controllers	Waterheaters	PV Systems	Storage Systems
(a) No TE	1594	1	1151	0	0
(b) Year 0	1594	755	1151	0	0
(c) Year 1	1594	755	1151	159	82
(d) Year 2	1594	755	1151	311	170
(e) Year 3	1594	755	1151	464	253

Simulation Architecture Model

The SGIP1 analysis, being a co-simulation, has a multiplicity of executables that are used to set-up the co-simulation, run the co-simulation, and process the data coming out of the co-simulation. The *Analysis Design Model* provides hints at which tools are used and how they interact but is not focused on how the tools fit together but rather how they can be used to achieve the necessary analysis objectives. This section fleshes out some of those details so that users are better able to understand the analysis process without having to resort to looking at the scripts, configuration files, and executable source code to understand the execution flow of the analysis.

Simulated Functionalities

The functionalities shown in Fig. 3.29 are implemented in simulation through a collection of software entities. Some of these entities perform dual roles (such as PYPOWER), solving equations that define the physical state of the system (in this case by solving the powerflow problem) and in also performing market operations to define prices (in this case by solving the optimal power flow problem).

- **GridLAB-D**
  - Simulates the physics of the electrical distribution system by solving the power flow of the specified distribution feeder model. To accomplish this it must provide the total distribution feeder load to PYPOWER (bulk power system simulator) and receives from it the substation input voltage.
  - Simulates the thermodynamics and HVAC thermostat control for all residential buildings in the specified distribution feeder model. Provides thermodynamic state information to the Substation Agent to allow formation of real-time energy bids.
  - Simulates the production of the solar PV panels and their local controller (for the cases that include such devices).
  - Simulates the physics of the energy storage devices and the behavior of their local controllers.
- **Substation Agent**
  - Contains all the transactive agents for the residential customers. Using the current state of the individual customers' residences (*e.g.* indoor air temperature) These agents form real-time energy bids for their respective customers and adjust HVAC thermostat setpoints based on the cleared price.

  - Aggregates all individual HVAC agents' real-time energy bids to form a single bid to present to the wholesale real-time energy market.

- **EnergyPlus**
  - Simulates the thermodynamics of a multi-zone structure (an elementary school in this case)
  - Simulates the integrated controller of said structure
  - Communicates electrical load of said structure to GridLAB-D for its use in solving the powerflow of the distribution feeder model.

- **PYPOWER**
  - After collecting the load information from GridLAB-D (and scaling it up to a value representative of an entire node in the transmission model) solves the bulk power system power flow to define the nodal voltages, communicating the appropriate value to GridLAB-D.
  - Using the bid information from the generation natively represented in the bulk power system model and the price-responsive load bids provided by the Substation Agent, find the real-time energy price for each node the bulk power system (the LMP) by solving the optimal power flow problem to find the least-cost dispatch for generation and flexible load. Communicate the appropriate LMP to the Substation Agent.

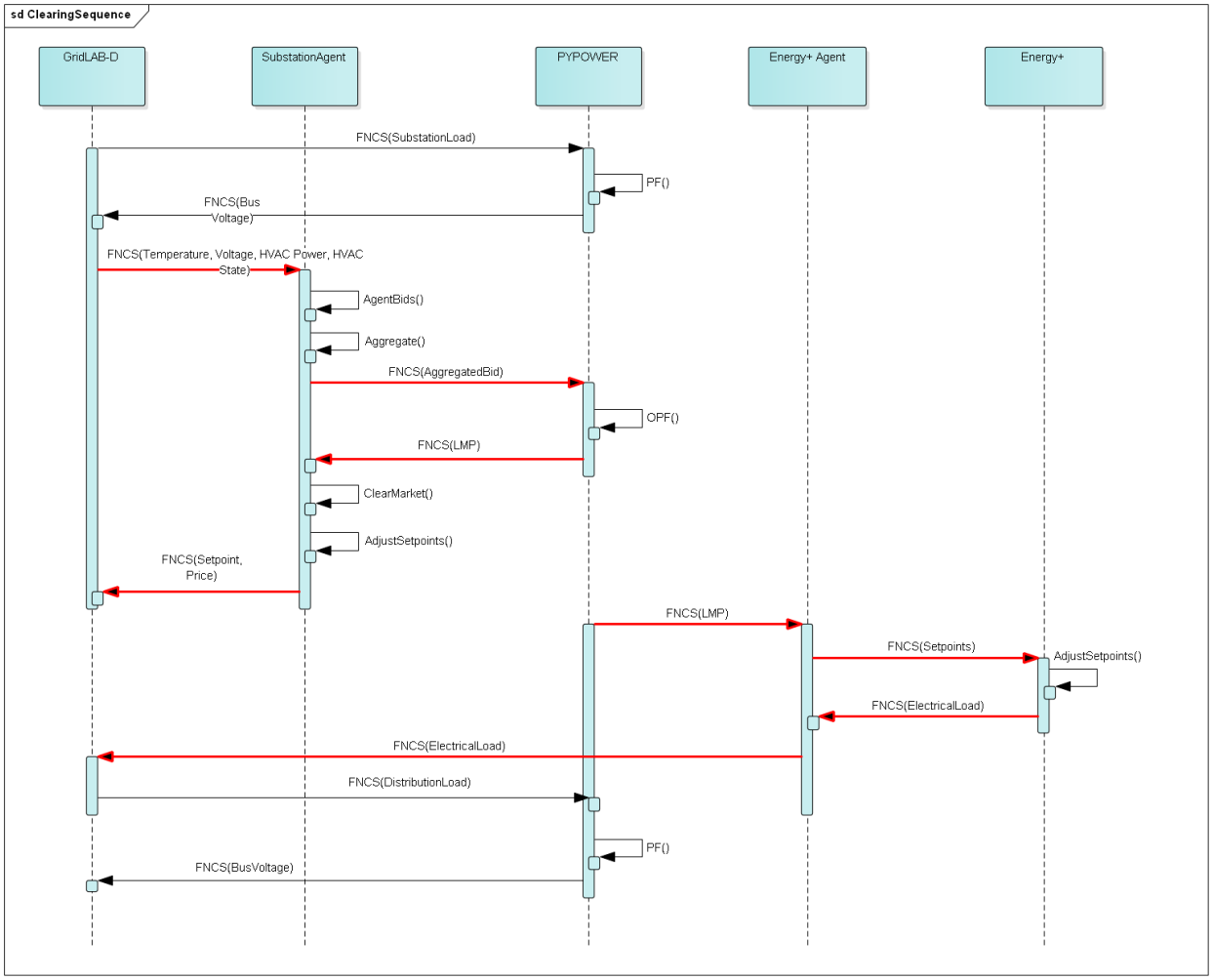


Fig. 3.33: Sequence of operations to clear market operations

Figure Fig. 3.33 is a sequence diagram showing the order of events and communication of information between the software entities.

Due to limitations in the load modeling provided by Energy+, some expected interactions are not included in this system model. Specifically:

- The loads modeled internally in Energy+ are not responsive to voltage and thus the interaction between it and GridLAB-D is only one way: Energy+ just provides a real power load; GridLAB-D does not assume a power factor and the the Energy Plus Agent (which is providing the value via FNCS) does not assume one either.
- The Energy Plus agent is only price responsive and does not provide a bid for real-time energy.

Software Execution

As is common in many analysis that utilize co-simulation, the SGIP1 analysis contains a relatively large number of executables, input, and output files. Though there are significant details in the *Analysis Design Model* showing the software components and some of the key data flows and interactions between them, it does not provide details of how the software is executed and interacts with each other. These details are provided below, focusing on the input and output files created and used by each executable.

Software Architecture Overview

Figure Fig. 3.34 provides the broadest view of the analysis execution. The central element is the "runSGIPn.sh" script which handles the launching of all individual co-simulation elements. To do this successfully, several input and configuration files need to be in place; some of these are distributed with the example and others are generated as a part of preparing for the analysis. Once the co-simulation is complete, two different post-processing scripts can be run to process and present that results.

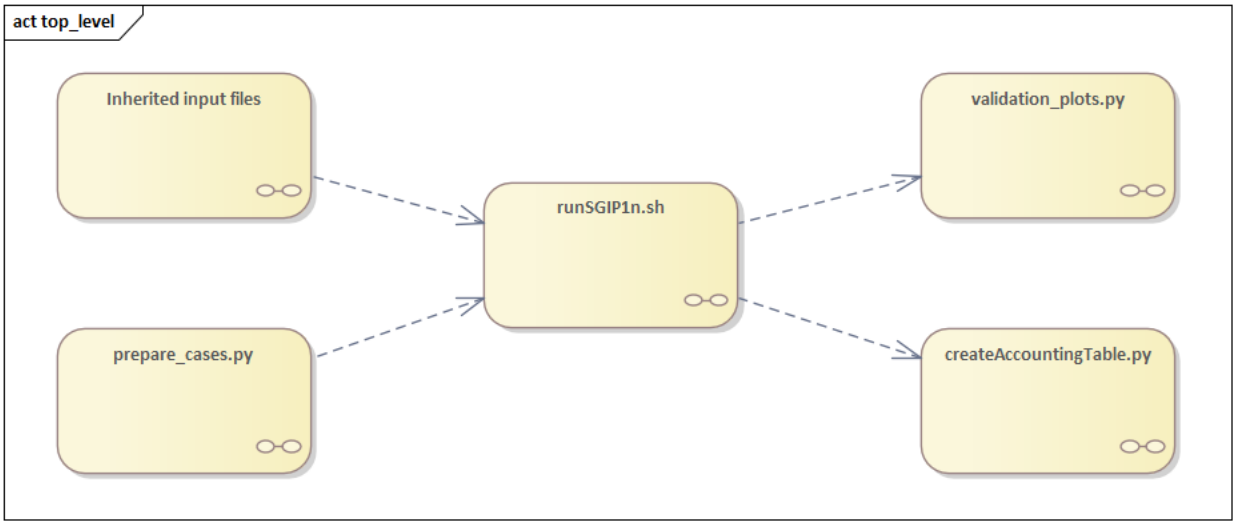


Fig. 3.34: Overview of the software execution path

Inherited Files

Figure Fig. 3.35 provides a simple list of files that are distributed with the analysis and are necessary inputs. The provenance of these files is not defined and thus this specific files should be treated as blessed for the purpose of the SGIP1 analysis.

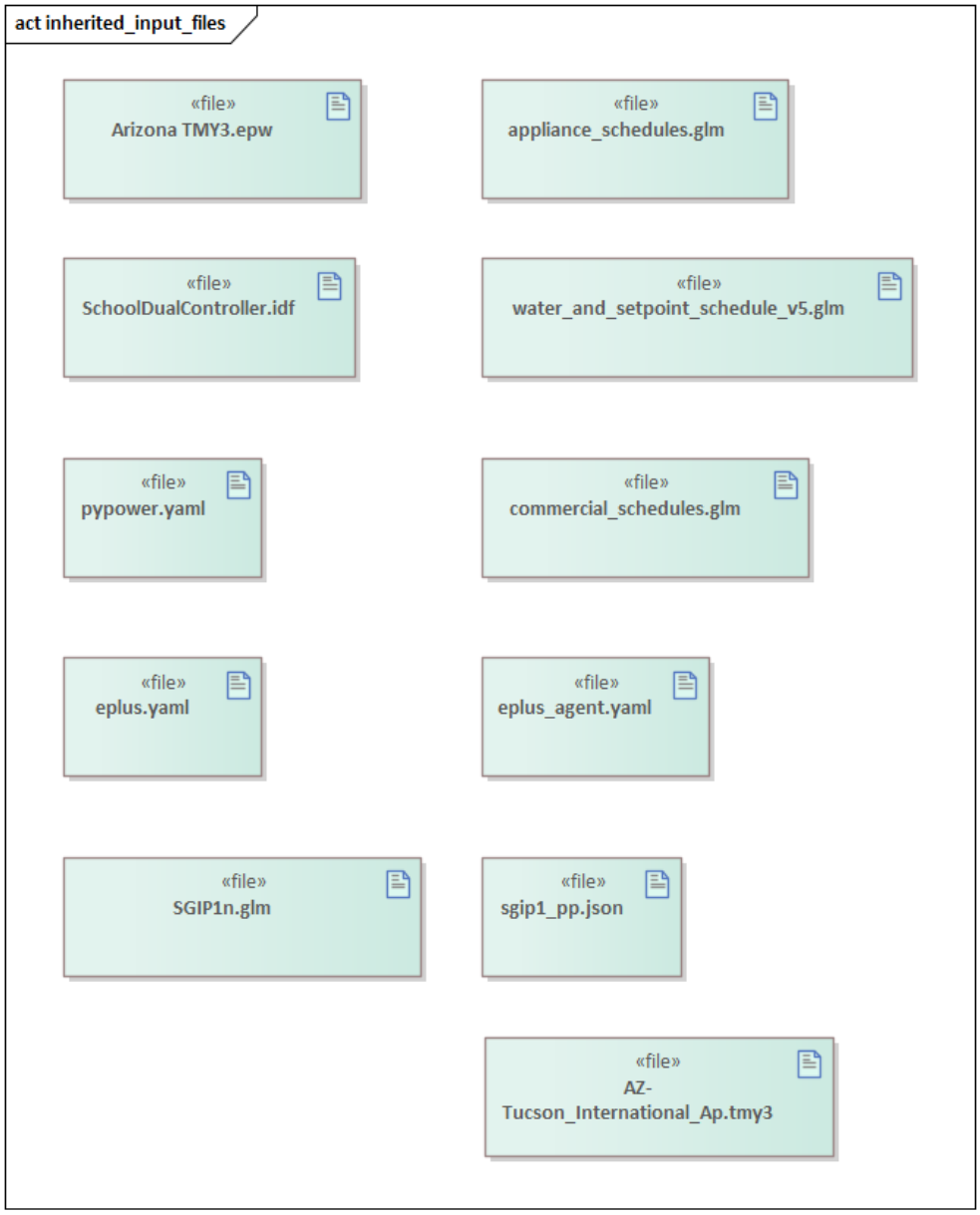


Fig. 3.35: List of files distributed with the SGIP1 analysis that are required inputs.

prepare_cases.py

Figure Fig. 3.36 shows the process by which the co-simulation-specific files are generated. The weather agent uses a specially-formatted weather file that is generated by the "weathercsv" method in "TMY3toCSV.py". After this completes the "glm_dict.py" script executes to create the GridLAB-D metadata JSON. Lastly, the "prep_substation.py" script runs to create co-simulation configuration files. "prepare_cases.py" does this for all the cases that the SGIP1 analysis supports.

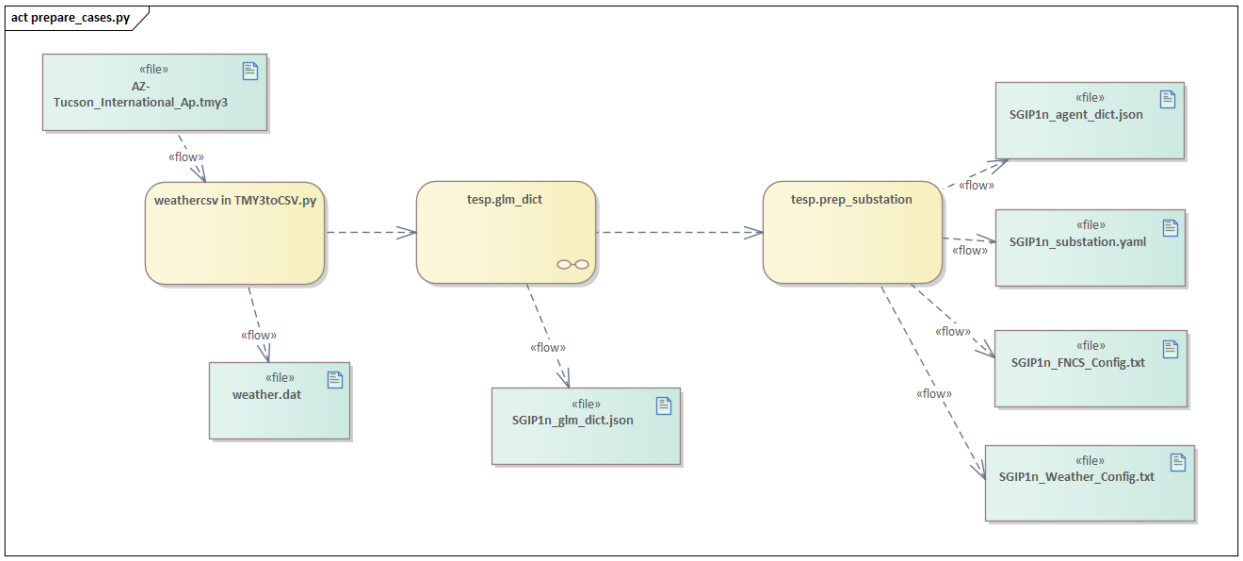


Fig. 3.36: Co-simulation files generated by "prepare_cases.py" in preparation of performing the analysis proper.

runSGIPn.sh

Figure Fig. 3.37 shows series of executables launched to run the SGIP1 co-simulation analysis. All of the activity blocks denote a specific executable with all being run in parallel to enable co-simulation. Each executable has its own set of inputs and outputs that are required and generated (respectively). Though most of these inputs are files (as denoted by the file icon), a few are parameters that are hard-coded into this script (*e.g.* the EnergyPlus Agent). Some input files have file dependencies of their own and these are shown as arrows without the "<<flow>>" tag. The outputs generated by each executable generally consist of a log file and any data collected in a metrics file.

validation_plot.py

Figure Fig. 3.38 shows the inputs files generated by the co-simulation that are used to generate plots used to validate the correct operation of the co-simulation. TESP provides scripts for post-processing the metrics files produced by the simulation tools and these are used to create Python dictionaries which can be manipulated to produce the validation plots.

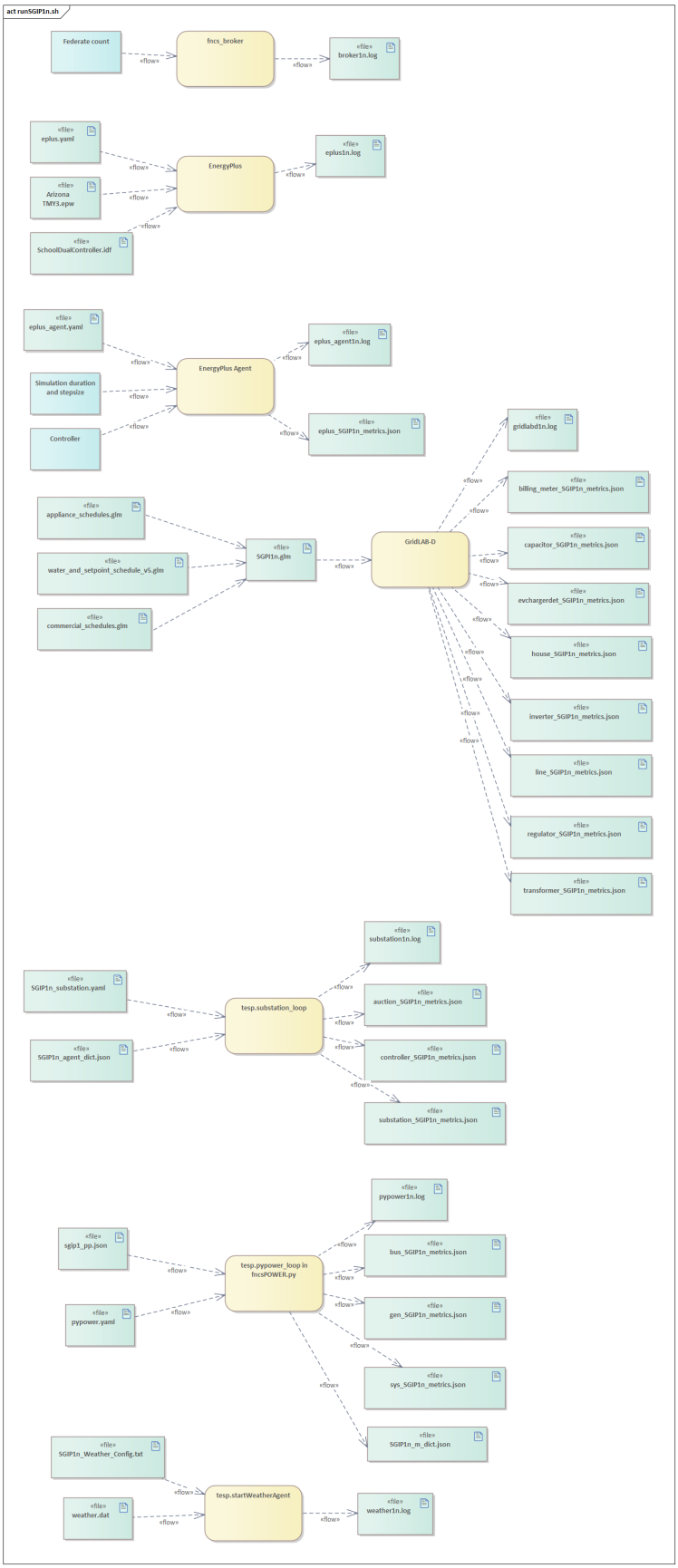


Fig. 3.37: Co-simulation executables launched by "runSGIP1n.sh" and their output metrics files

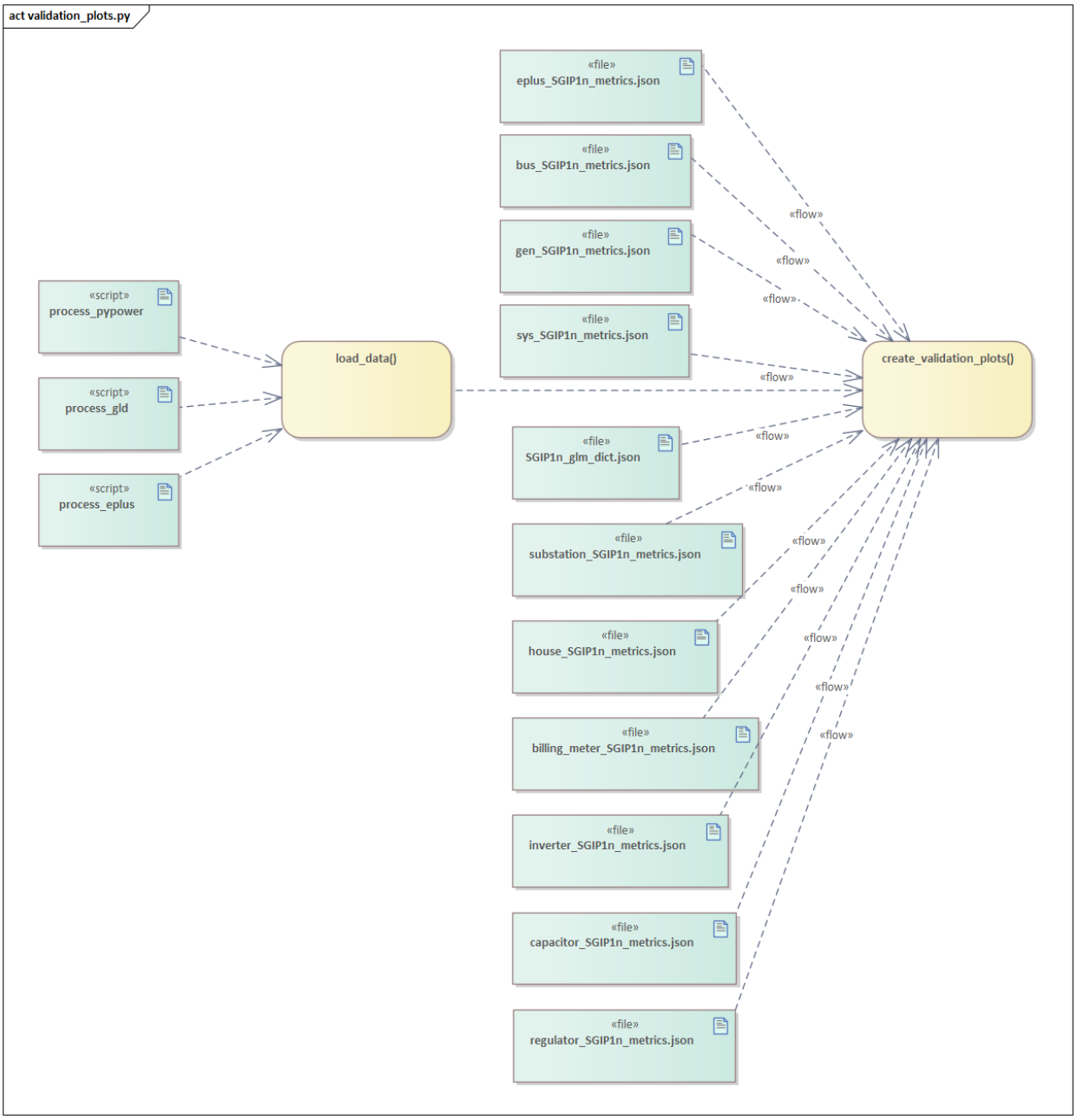


Fig. 3.38: Post-processing of the output metrics files to produce plots to validate the correct execution of the co-simulation.

createAccountingTable.py

Figure Fig. 3.39 shows the input metrics files to used to calculate the final accounting table output from the metrics identified by the *Valuation Model*.

Data Collection

The data collection for TESP is handled in a largely standardized way. Each simulation tool produces an output dataset with key measurements. This data is typically stored in a JSON file (with an exception or two where the datasets are large and HDF5 is used). The specific data collected is defined in the *metrics section* of the TESP *Design Reference*.

The JSON data files are post-processed by Python scripts (one per simulation tool) to produce Python dictionaries that can then be queried to further post-process the data or used directly to create graphs, charts, tables or other presentations of the data from the analysis. Metadata files describing the models used in the analysis are also used when creating these presentations.

Running the Example

As shown in Table 3.4, the SGIP1 example is actually a set of five separate co-simulation runs. Performing each run takes somewhere around two hours (depending on the hardware) though they are entirely independent and thus can be run in parallel if sufficient computation resources are available. To avoid slowdowns due to swapping, it is recommended that each run be allocated 16Gb of memory.

To launch one of these runs, only a few simple commands are needed:

```
cd ~/tesp/examples/sgip1
python3 prepare_cases.py # Prepares all SGIP1 cases
# run and plot one of the cases
./runSGIP1b.sh
```

`./runSGIP1b.sh` will return a command prompt with the co-simulation running in the background. To check how far along the co-simulation monitoring one of the output files is the most straight-forward way:

```
tail -f SGIP1b.csv
```

The first entry in every line of the file is the number of seconds in the co-simulation that have been completed thus far. The co-simulation is finished at 172800 seconds. After that is complete, a set of summary plots can be created with the following command:

```
python3 plots.py SGIP1b
```

Analysis Results - Model Validation

The graphs below were created by running `validation_plots.py` (**TODO:** Update default path to match where the data will be) to validate the performance of the models in the co-simulation. Most of these plots involve comparisons across the cases evaluated in this study (see Table 3.4).

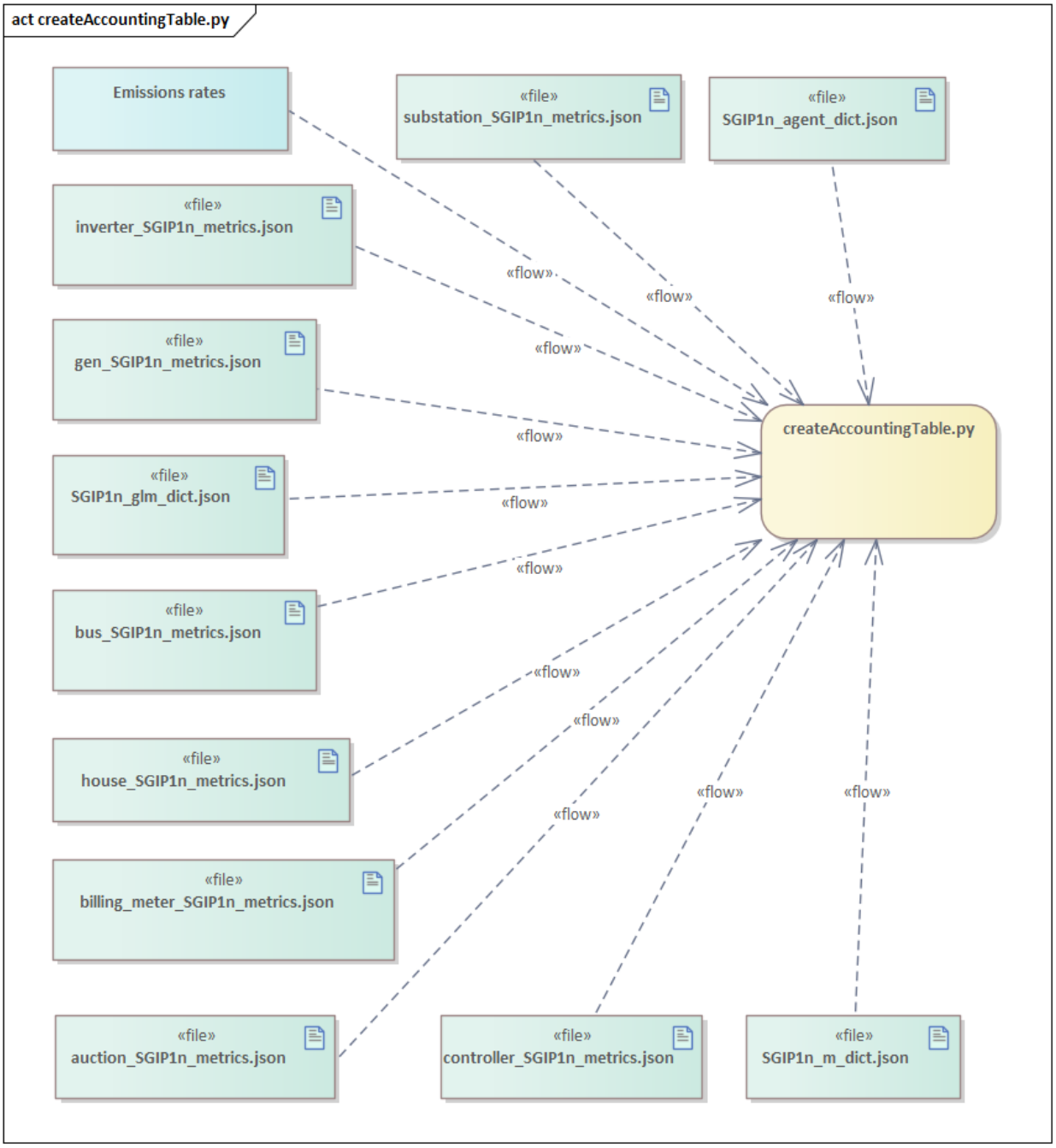


Fig. 3.39: Post-processing of the output metrics files to produce the necessary metrics for the accounting table.

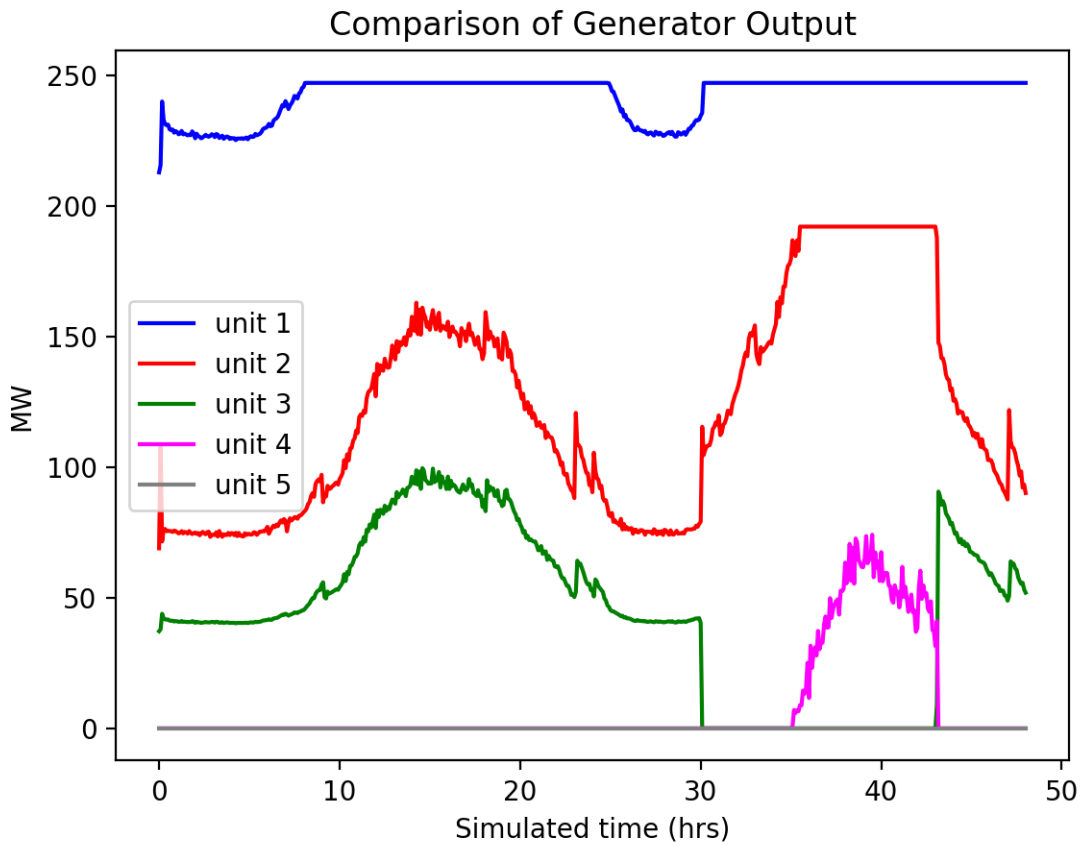


Fig. 3.40: Generator outputs of bulk power system, showing the loss of Unit 3 on the second day.

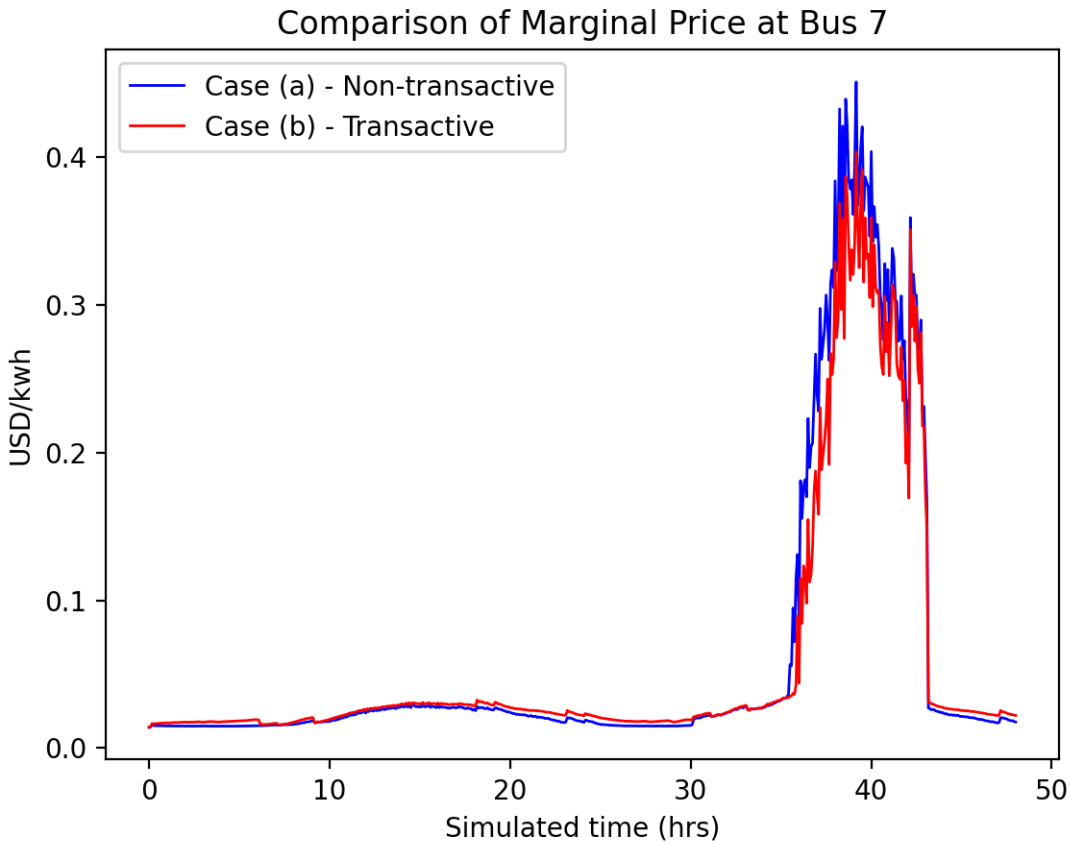


Fig. 3.41: Wholesale market prices (LMPs) for base and transactive cases, showing lower prices during the peak of the day as transactively participating loads respond.

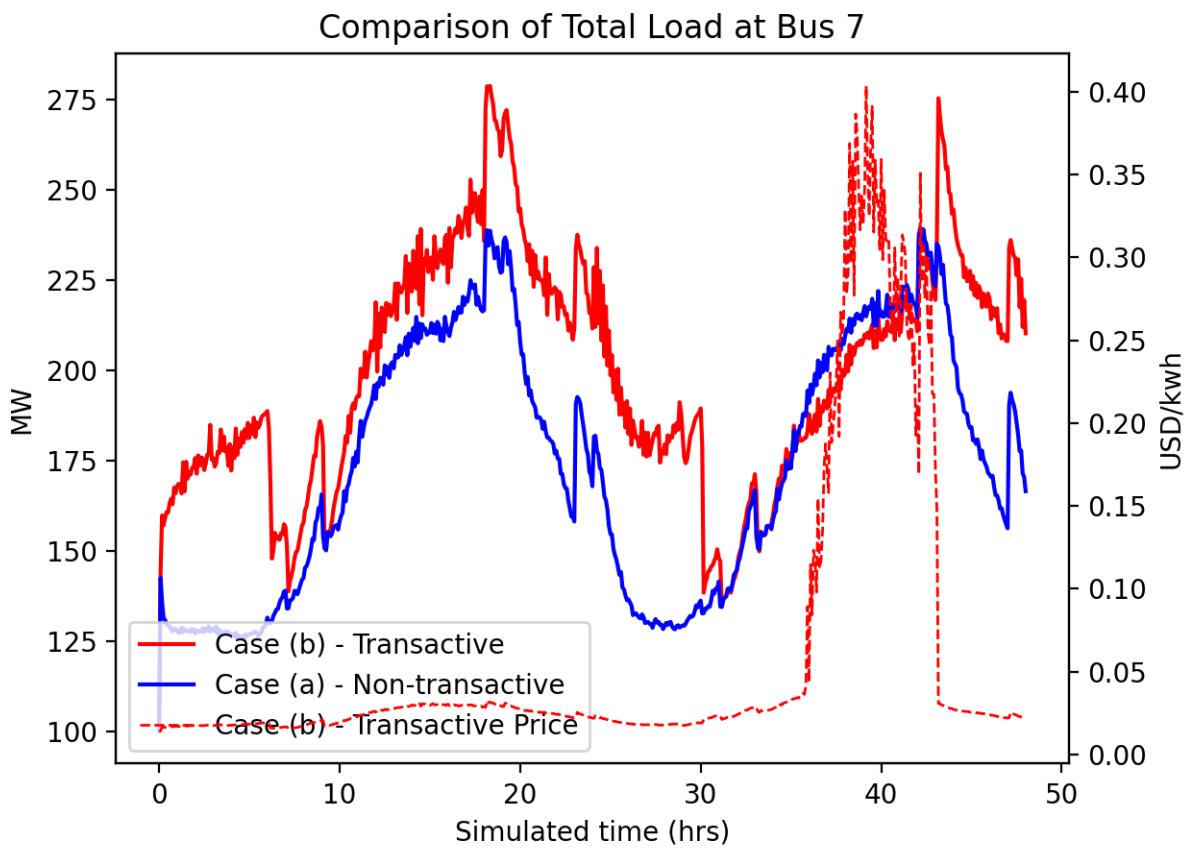


Fig. 3.42: Total load for transactive feeder in base and transactive case. Should show peak-shaving, valley-filling, and snapback as prices come down off their peak.

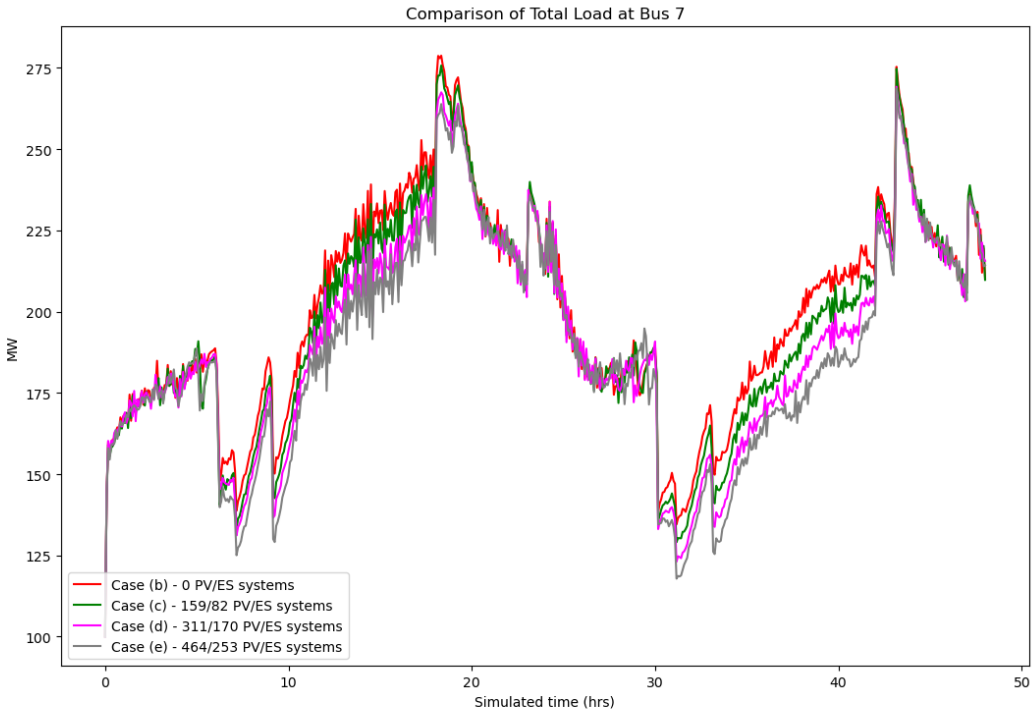


Fig. 3.43: Total load for transactive feeder in for four transactive cases with increasing levels of rooftop solar PV and energy storage penetration.

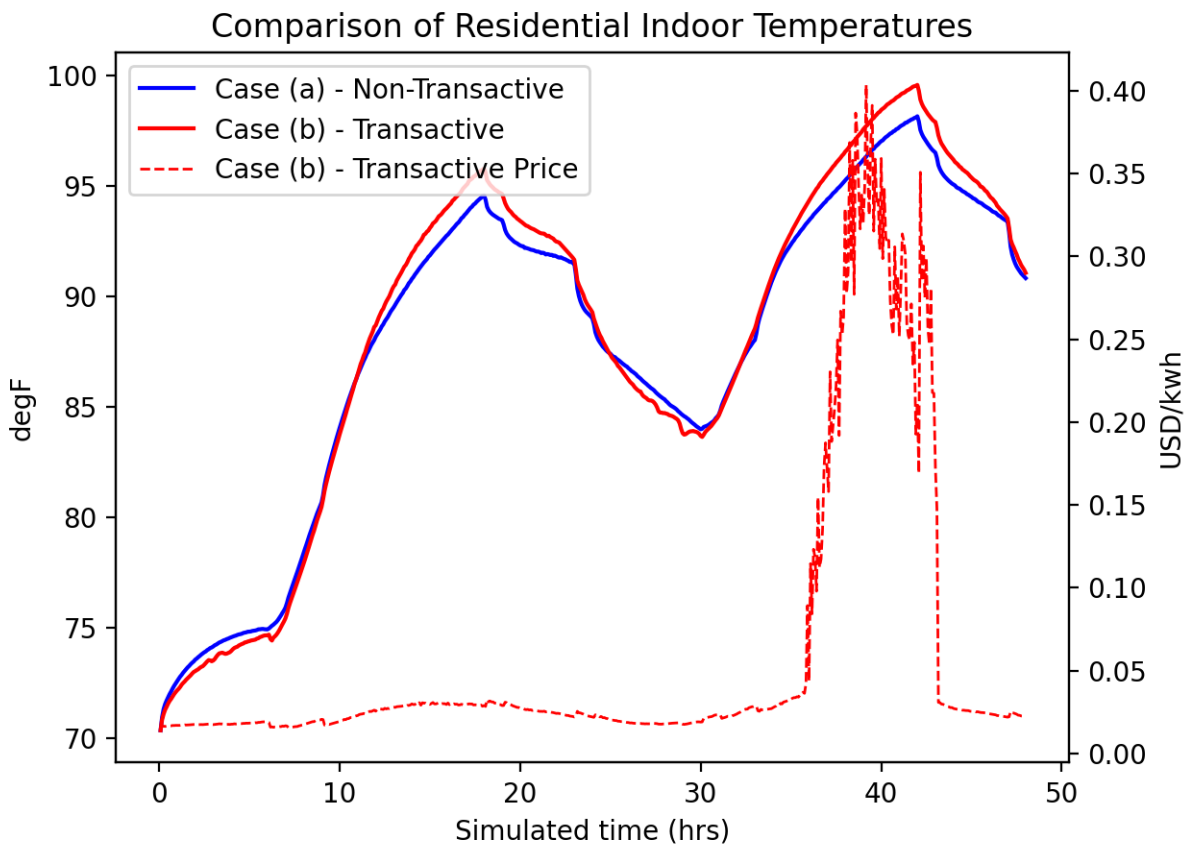


Fig. 3.44: Average residential indoor air temperature for all houses in both base and transactive case. The effect of the transactive controller for the HVACS drives lower relatively lower temperatures during low price periods and relatively higher prices during higher periods.

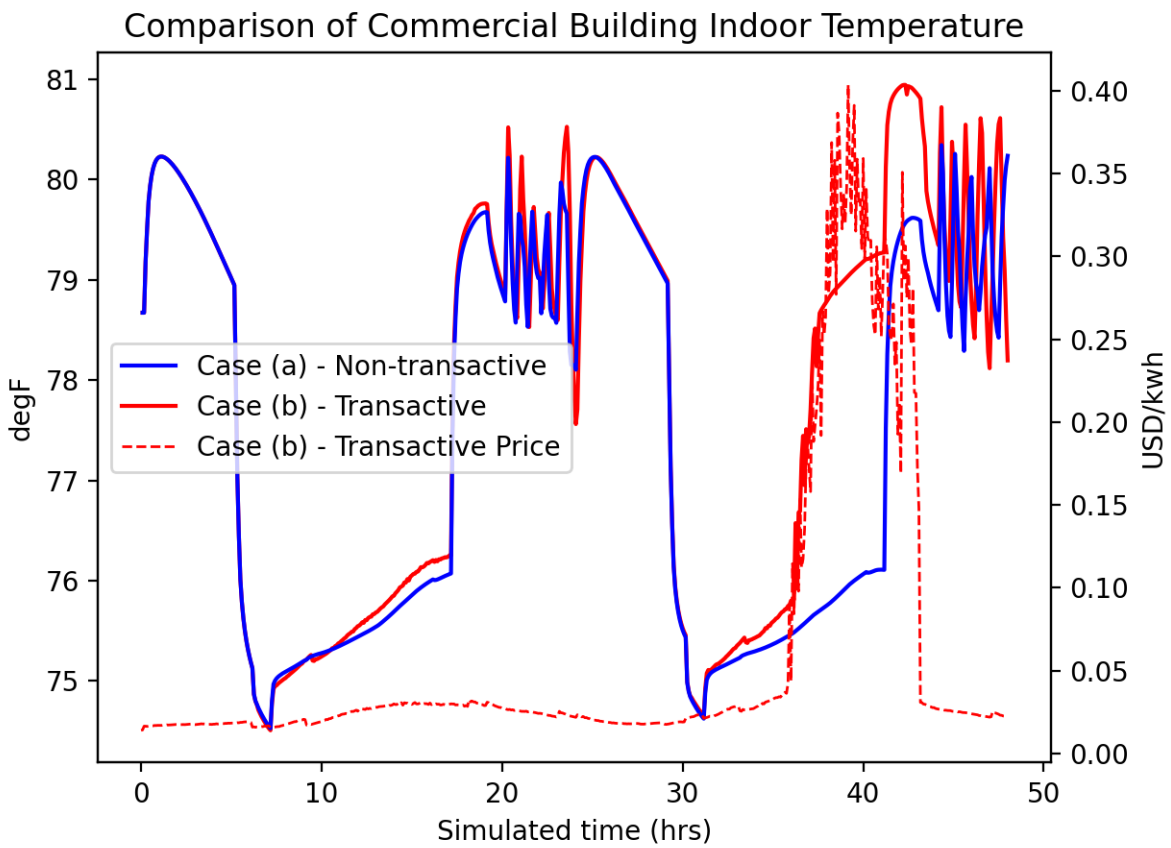


Fig. 3.45: Commercial building (as modeled in Energy+) indoor air temperature for the base and transactive case. Results should be similar to the residential indoor air temperature with lower temperatures during low-price periods and higher temperatures during high-price periods.

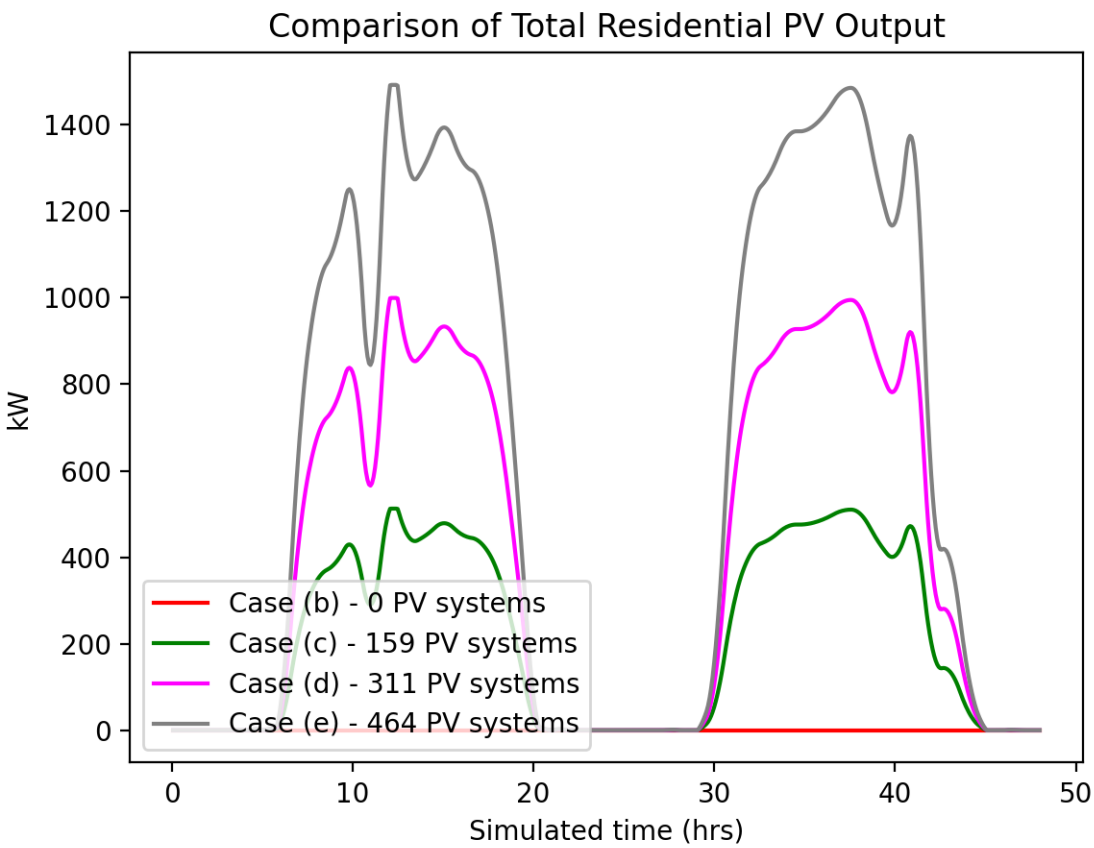


Fig. 3.46: Total residential rooftop solar output on the transactive feeder across the four cases within increasing penetration. The rooftop solar is not price responsive. As expected, increasing PV penetration showing increased PV production.

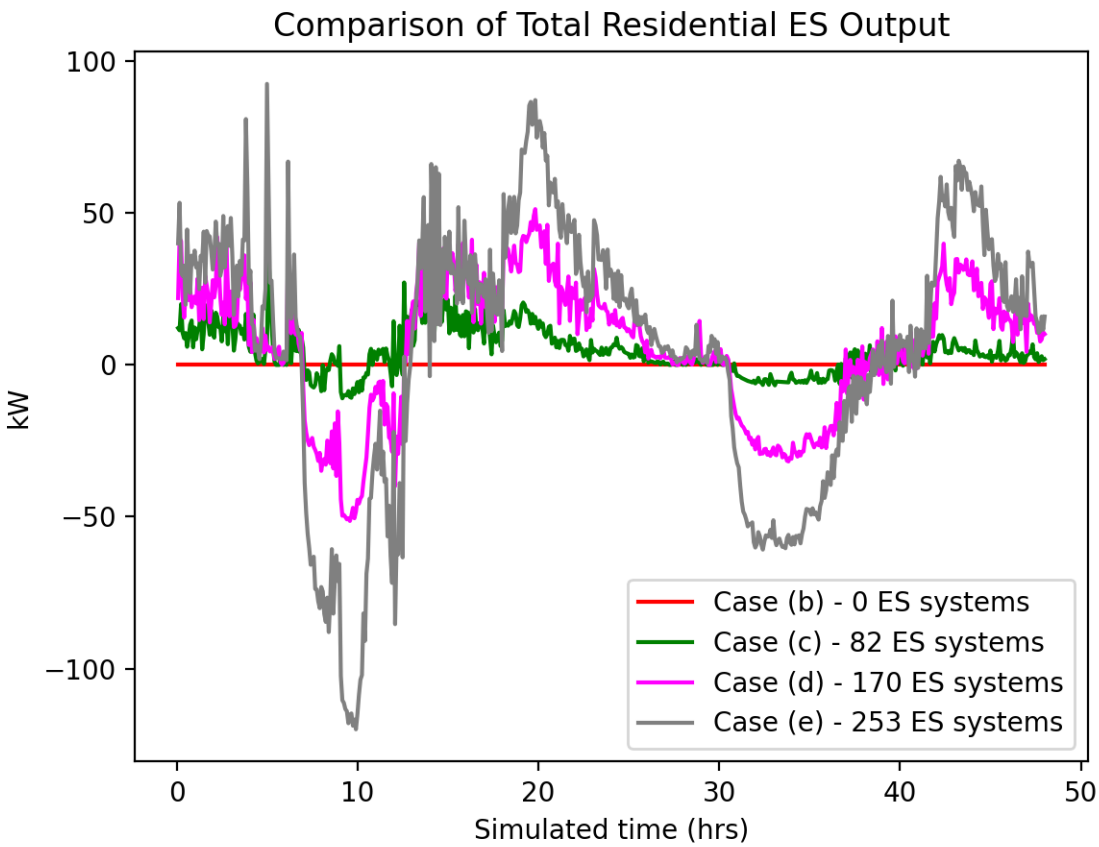


Fig. 3.47: Total residential energy storage output on the transactive feeder across the four cases within increasing penetration. The energy storage controller engages in peak-shaving and valley-filling based on the billing meter for the residential customer.

Analysis Results - Key Performance Metrics

The final results for the key performance metrics are presented below in Table 3.5 and Table 3.6. These tables are provided to help benchmark results.

Table 3.5: Accounting Table, Day 1

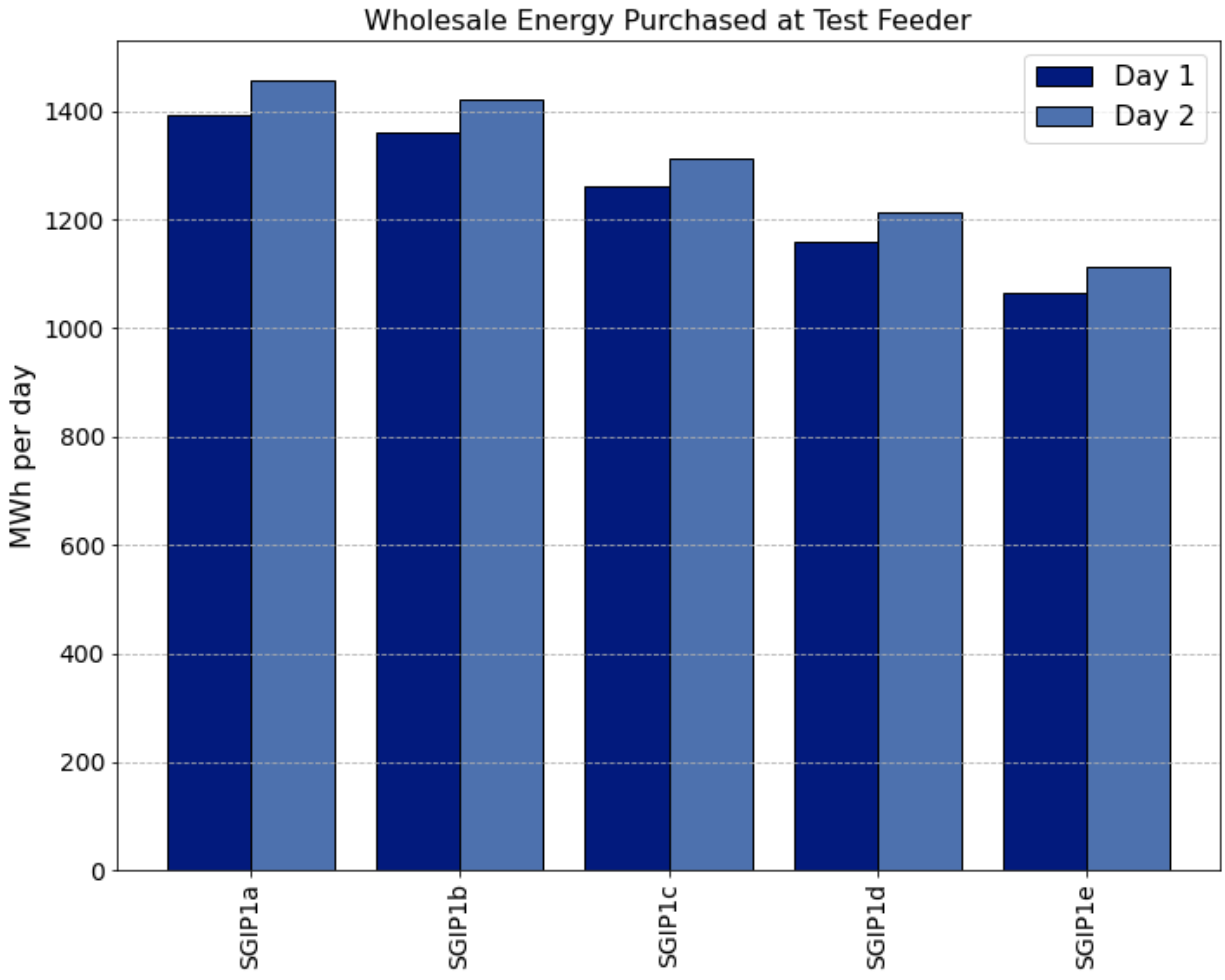
Metric Description	SGIP1a Day 1	SGIP1b Day 1	SGIP1c Day 1	SGIP1d Day 1	SGIP1e Day 1
Wholesale electricity purchases for test feeder (MWh/d)	1394	1363	1261	1159	1065
Wholesale electricity purchase cost for test feeder ($/day)	$31,414.83	$33,992.28	$30,940.02	$27,869.27	$25,287.68
Total wholesale generation revenue ($/day)	$213,441.28	$237,176.68	$230,705.91	$224,178.12	$218,903.29
Transmission and Distribution Losses (% of MWh generated)	0.03	0.03	0.03	0.03	0.03
Average PV energy transacted (kWh/day)	0.0	0.0	17.6	34.3	51.2
Average PV energy revenue ($/day)	$0.00	$0.00	$127.65	$242.23	$353.86
Average ES energy transacted (kWh/day)	0.00	0.00	0.68	0.98	0.88
Average ES energy net revenue	$0.00	$0.00	$4.98	$7.84	$8.16
Total CO2 emissions (MT/day)	0.70	0.79	0.78	0.76	0.75
Total SOx emissions (kg/day)	0.01	0.01	0.01	0.01	0.01
Total NOx emissions (kg/day)	0.05	0.05	0.05	0.05	0.05

Table 3.6: Accounting Table, Day 2

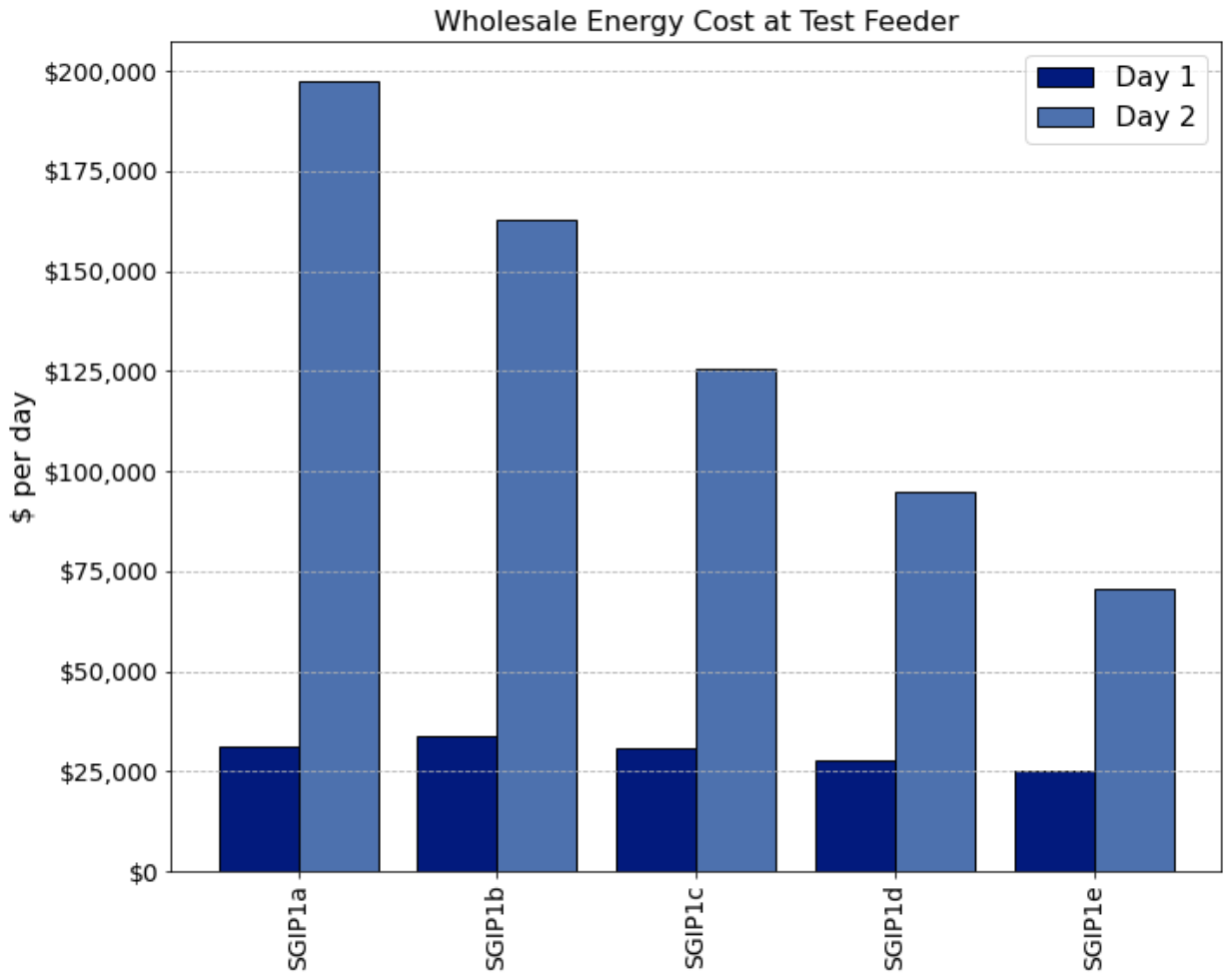
Metric Description	SGIP1a Day 2	SGIP1b Day 2	SGIP1c Day 2	SGIP1d Day 2	SGIP1e Day 2
Wholesale electricity purchases for test feeder (MWh/d)	1458	1421	1314	1213	1112
Wholesale electricity purchase cost for test feeder ($/day)	$197,668.90	$162,838.08	$125,429.86	$95,077.07	$70,833.33
Total wholesale generation revenue ($/day)	$1,219,691.4	$1,065,540.5	$884,707.64	$724,209.63	$581,815.57
Transmission and Distribution Losses (% of MWh generated)	0.03	0.03	0.03	0.03	0.03
Average PV energy transacted (kWh/day)	0.0	0.0	18.5	36.0	53.8
Average PV energy revenue ($/day)	$0.00	$0.00	$667.30	$1,034.39	$1,188.40
Average ES energy transacted (kWh/day)	0.00	0.00	0.08	0.09	0.02
Average ES energy net revenue	$0.00	$0.00	$4.77	$8.56	$11.76
Total CO2 emissions (MT/day)	3.61	3.21	2.72	2.27	1.86
Total SOx emissions (kg/day)	0.03	0.03	0.02	0.02	0.02
Total NOx emissions (kg/day)	0.23	0.21	0.17	0.15	0.12

The following graphs were created by running `createAccountingTable.py` and `plotAccountingTable.py`, which calls the function `lca_standard_graphs.py`. These plots involve comparisons across the cases evaluated in this study (see Table 3.5 and Table 3.6). As shown in :numref: *tbl_sgip1*, the cases a through e correspond to a growth model of progressive adoption of transactive energy. Case a is no transactive energy and case b is the control year with transactive. Cases c through e account for increased adoption of PV and energy storage systems. These cases are analyzed over two days of operation, where the first day is a control with regular operations, and the second experiences the trip of a generator. These graphs convey the respective results from the simulation.

The wholesale energy purchased at the test feeder reduces with the growth model, however with the generator trip on day two, energy demand increases slightly.


Wholesale Energy Purchased at Test Feeder
Day 1
Day 2
MWh per day
0
200
400
600
800
1000
1200
1400
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e

The cost of wholesale energy decreases slightly on day one with adoption of PV and energy storage. This decrease in cost is more dramatic on the second day with the generator trip.



The total revenue to the generators is much more on the second day with the generator failure, although this revenue reduces with the growth model.

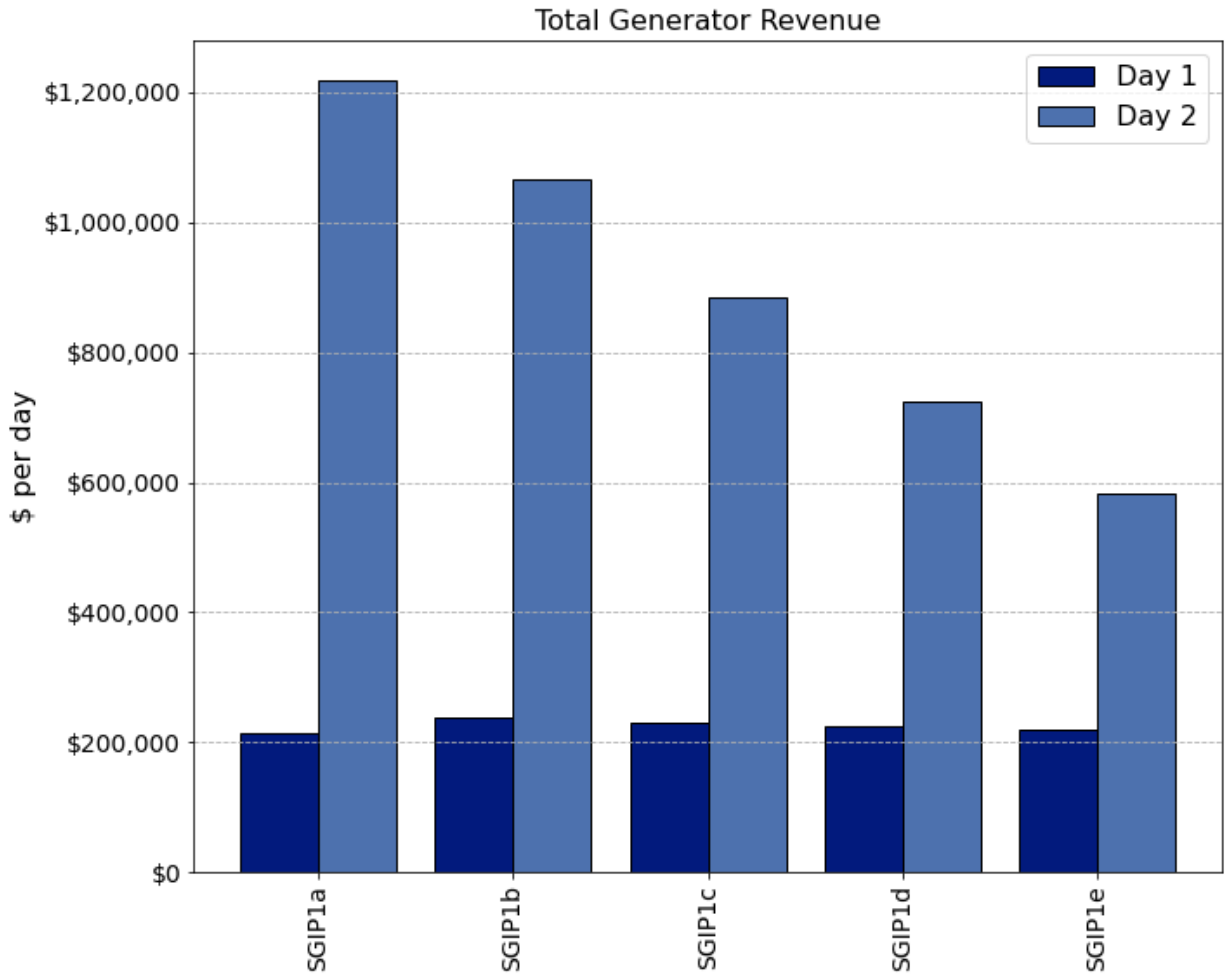
The average PV energy transacted in the system increases with adoption, and this amount progressively diverges with the generator trip on the second day.

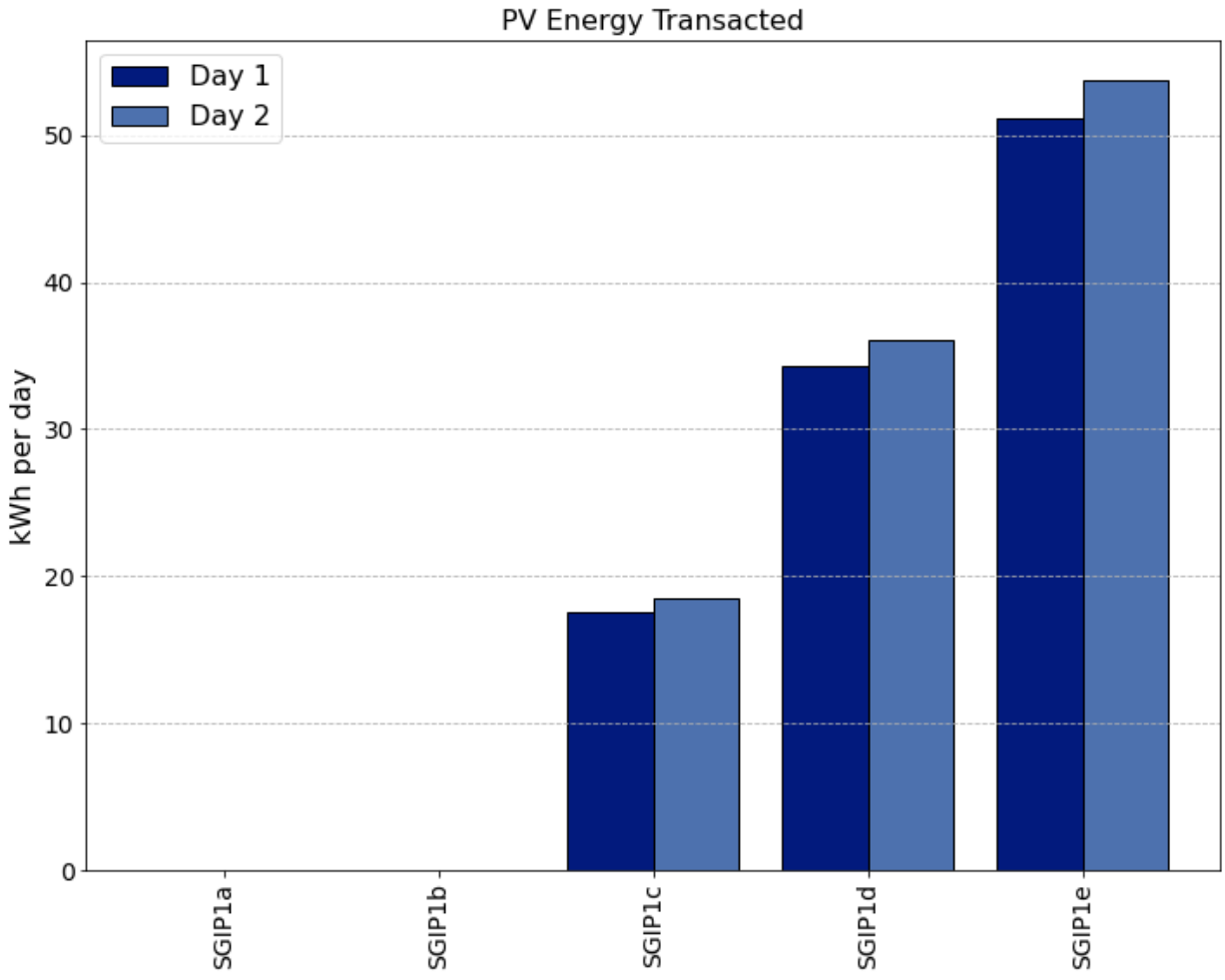
The average revenue to households with PV markedly increases on the second day with the generator trip.

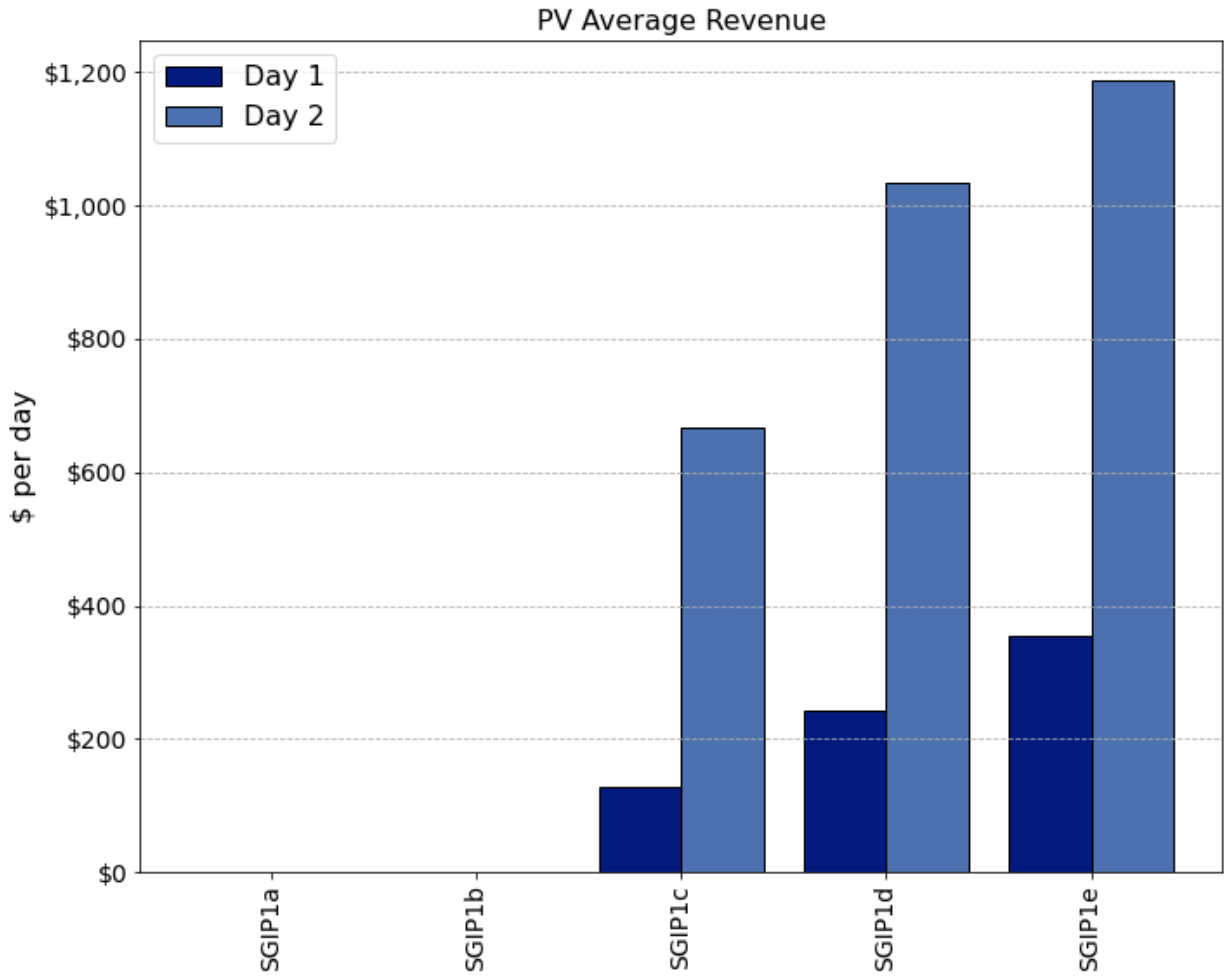
The average energy storage energy transacted is much larger on day one compared with day two. The impact of the generator trip is nonlinear with increasing adoption of storage.

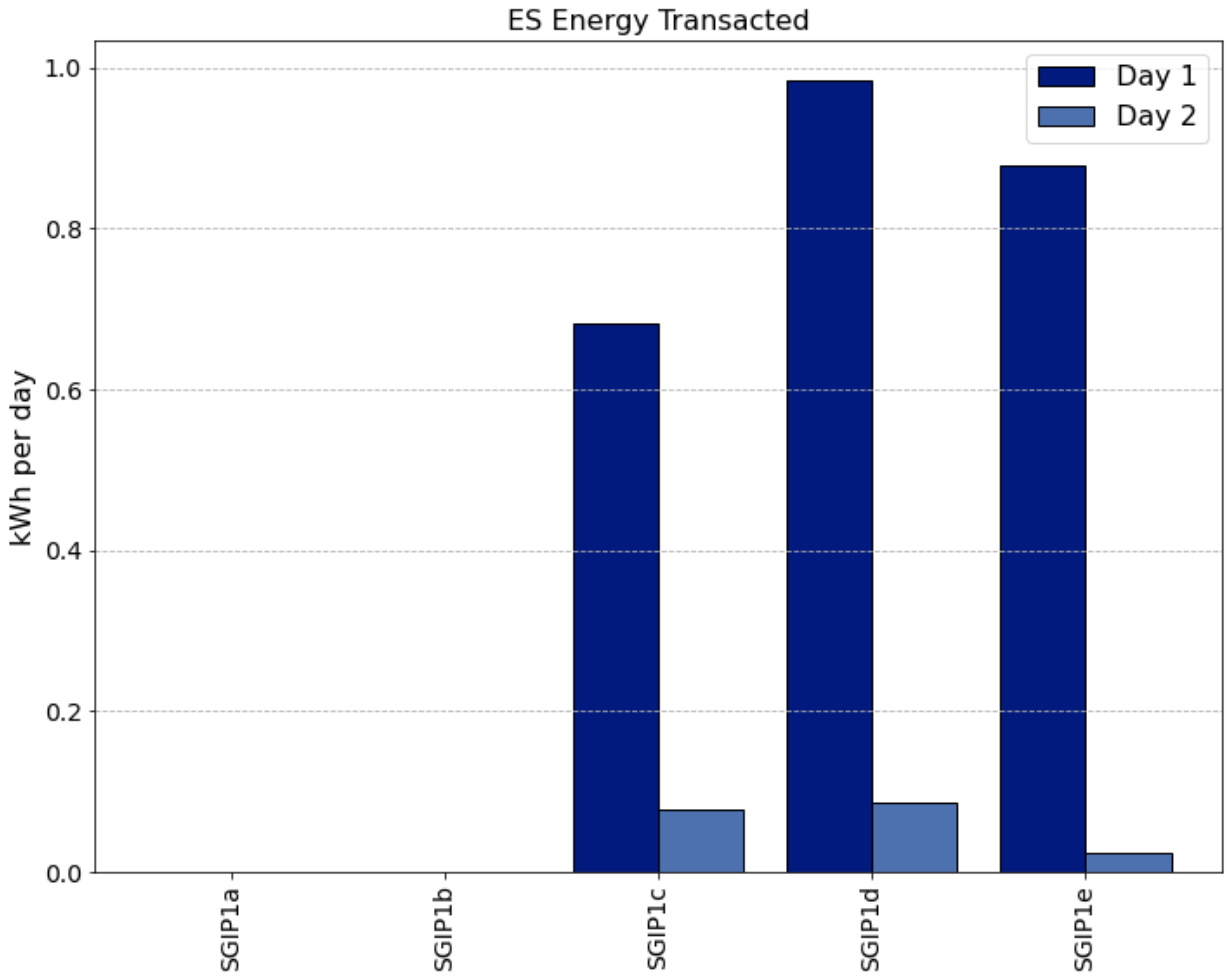
Regardless of the average energy transacted, the average revenue to households is much larger on day two.

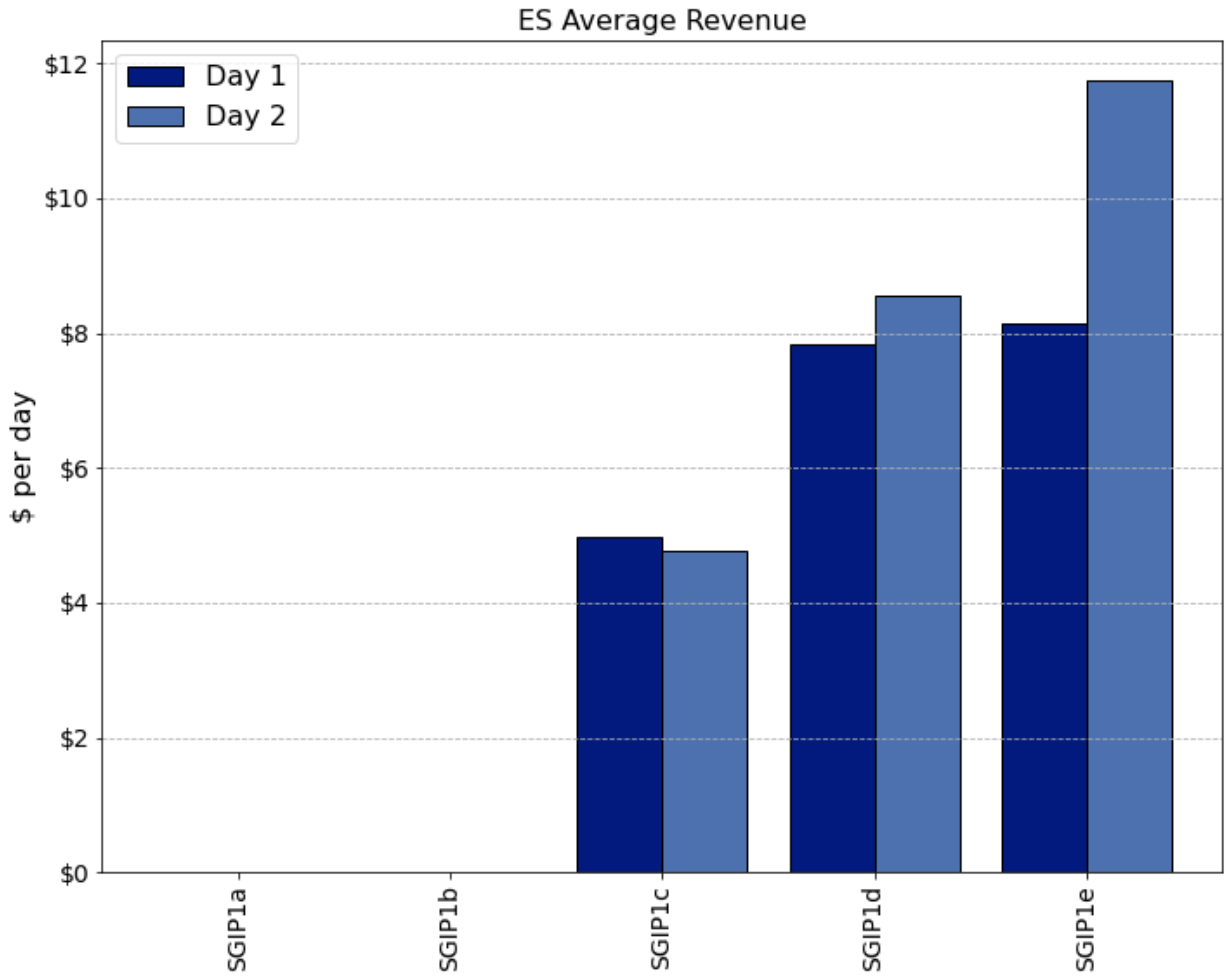
The following graph dispays the total daily emissions by greenhouse gas type (CO2 is in units of MT, or 1000 kg). Between case a (no transactive) and case b (transactive year 0), the CO2 emissions jump by 100 MT in day one. The emissions in day two increase between these two cases as well by about half as much. Across all cases, the total emissions on day one are higher with transactive compared to without. On day two, the growth model cases experience less total emissions than the non-transactive case.

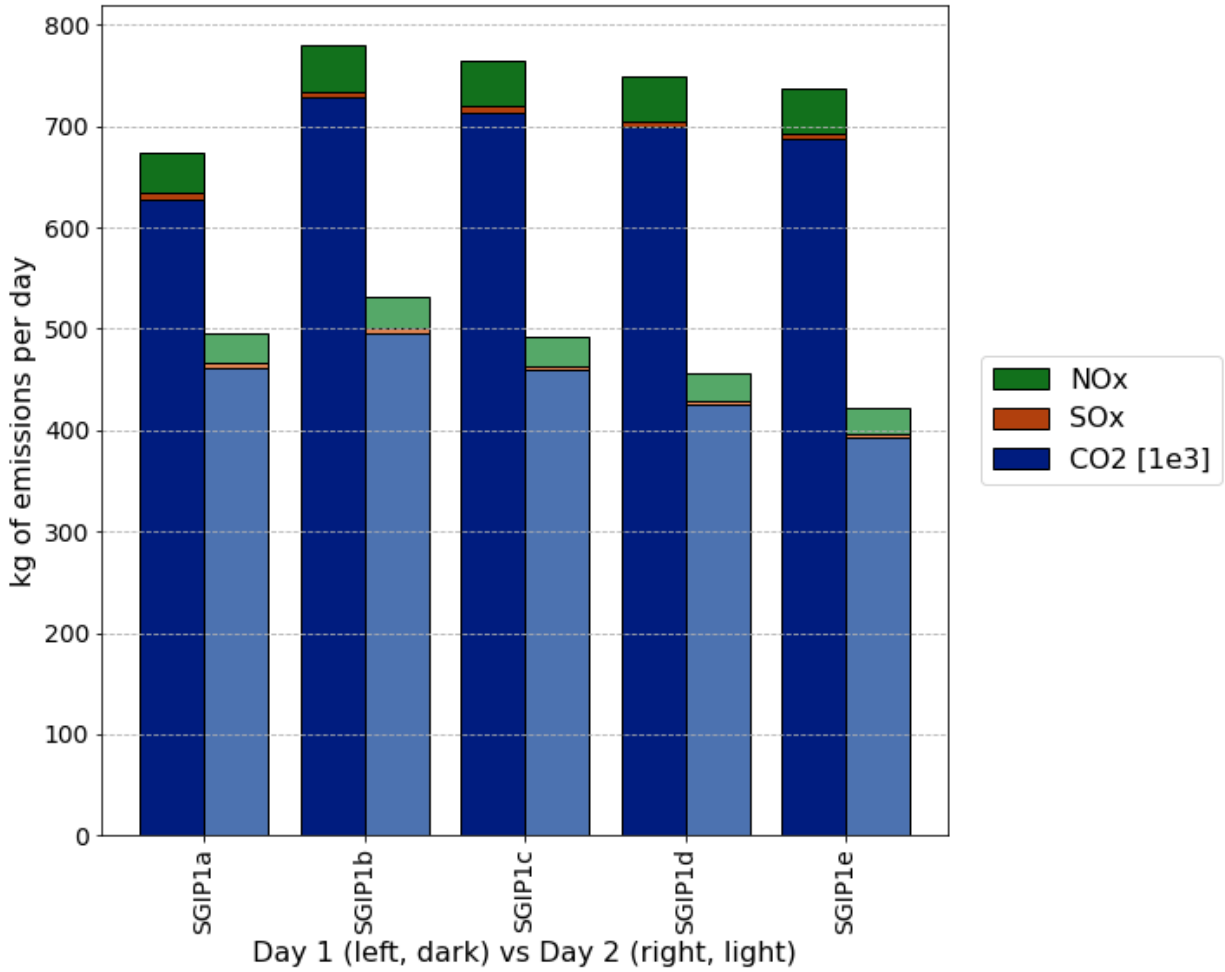

Total Generator Revenue
Day 1
Day 2
$ per day
$1,200,000
$1,000,000
$800,000
$600,000
$400,000
$200,000
$0
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e


PV Energy Transacted
Day 1
Day 2
kWh per day
0
10
20
30
40
50
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e


PV Average Revenue
Day 1
Day 2
$1,200
$1,000
$800
$600
$400
$200
$0
$ per day
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e


ES Energy Transacted
kWh per day
1.0
0.8
0.6
0.4
0.2
0.0
Day 1
Day 2
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e


ES Average Revenue
Day 1
Day 2
$ per day
$12
$10
$8
$6
$4
$2
$0
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e


800
700
600
500
400
300
200
100
0
kg of emissions per day
SGIP1a
SGIP1b
SGIP1c
SGIP1d
SGIP1e
Day 1 (left, dark) vs Day 2 (right, light)
NOx
SOx
CO2 [1e3]

Related Publications

This use of TESP to perform the SGIP1 analysis resulted in the following related publications:

S. E. Widergren, D. J. Hammerstrom, Q. Huang, K. Kalsi, J. Lian, A. Makhmalbaf, T. E. McDermott, D. Sivaraman, Y. Tang, A. Veeramany, and J. C. Woodward. Transactive Systems Simulation and Valuation Platform Trial Analysis. Technical Report PNNL-26409, Pacific Northwest National Laboratory (PNNL), Richland, WA (United States), Richland, WA, Apr. 2017. DOI: 10.2172/1379448. Available at: http://www.osti.gov/servlets/purl/1379448/

Q. Huang, T. McDermott, Y. Tang, A. Makhmalbaf, D. Hammerstrom, A. Fisher, L. Marinovici, and T. D. Hardy. Simulation-Based Valuation of Transactive Energy Systems. Power Systems, IEEE Transactions on, May 2018. DOI: 10.1109/TPWRS.2018.2838111. Available at: https://ieeexplore.ieee.org/document/8360969/

3.2.2 Distribution System Operator and Transactive (DSO+T) Study

The Pacific Northwest National Laboratory completed a multi-year study where a transactive energy system was implemented to allow a number of distribution system assets to participate in an integrated retail and wholesale real-time and day-ahead energy markets. The study had the following specific objectives

- Does the large-scale transactive coordination produce stable, effective control?
- Is a DSO+T deployment of flexible assets at-scale cost effective, particularly comparing batteries vs flexible loads and deployments in moderate- vs high-renewable power systems?
- How much could a DSO+T implementation benefit and save consumers, looking at comparisons between those that participate in the transactive system and those that don't and residential versus commercial customers.

For this study, an ERCOT-spanning electrical power system model was constructed that reached in scope from bulk power system generation to individual customer loads. This included modeling 10,000s of thousands of customers with unique residential, load, and market-participating device models, as shown in Figure :numref: *fig_dsot_scope_scale*.

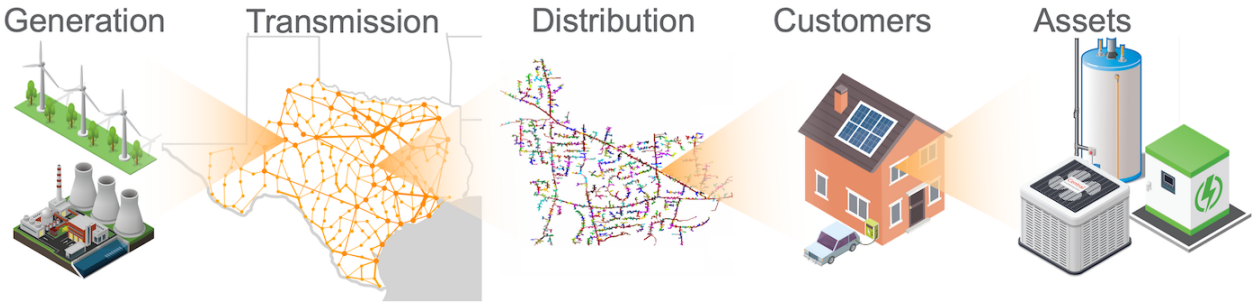


Fig. 3.48: DSO+T modeled infrastructure

Some fraction of the customer-owned assets (HVAC, electric water heaters, and batteries) were implemented such that they participated in a retail energy market run by the modeled distribution system operators (DSOs). These DSOs aggregated the price-responsiveness of the assets and presented this flexibility to the wholesale market being run by the transmission system operator (TSO). Both entities ran real-time and day-ahead energy markets.

The simulations were run on models not only replicating the state of the ERCOT power system today but also considering a number of alternative futures scenarios. Specifically, a high-renewable generation mix with increased utility-scale wind and both increased utility-scale and distributed rooftop solar was modeled. On the load side, the transactive mechanism was compared using just flexible loads as well as just using customer-owned batteries.

An entire calendar year was simulated to capture the effects of peak loads that may occur in summar or winter, depending on the geographic location within the system.

The documentation of the system provided here is a summary of the extensive documentation produced by the DSO+T analysis team. The version of codebase included here in TESP is similar to but not identical to that used to perform the study. Results produced by this codebase will likewise be similar but not identical. Comprehensive documentation of the study can be found in the following reports:

- Distribution System Operator with Transactive (DSO+T) Study Volume 1: Main Report.
- DSO+T: Integrated System Simulation DSO+T Study: Volume 2
- DSO+T: Transactive Energy Coordination Framework DSO+T Study: Volume 3
- DSO+T: Valuation Methodology and Economic Metrics DSO+T Study: Volume 4

Software Architecture

The DSO+T analysis, though run on a single local compute node, has a relatively complex software architecture. There are a number of software entities that interact through a variety of means, as shown in Figure Fig. 3.49.

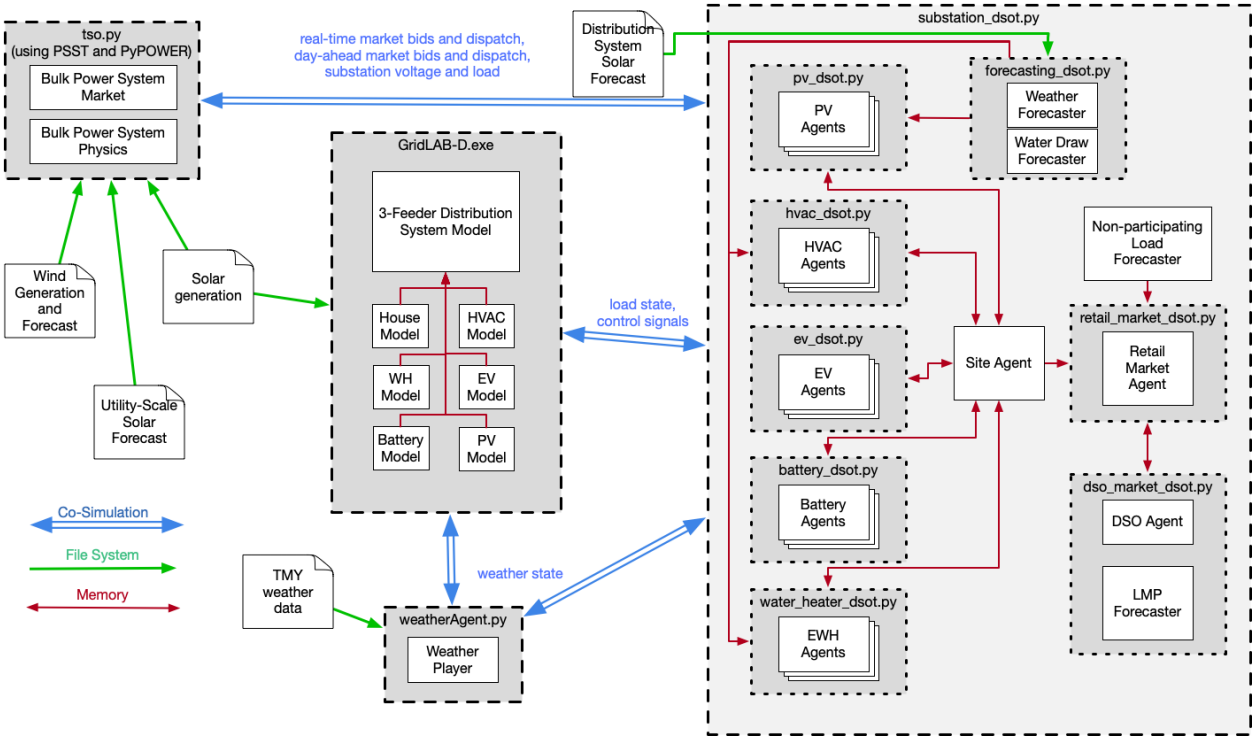


Fig. 3.49: High-level DSO+T software architecture diagram

Each of the gray dashed-outlined boxes represents a key executable in performing the DSO+T analysis. Some of these executables use other software entities from the same codebase to perform specific functions. For example, GridLAB-D has specific function written to allow it to implement models of water heaters, HVAC systems, and residential structures. Similarly, the substation_dsot.py calls other Python scripts to implement control agents for the loads modeled in GridLAB-D. These interactions are shown in dark-red arrows to indicate the data exchange between the software entities is happening in-memory and is largely invisible to the analyst running the simulation.

Another form of data-exchange is realized through simple file-system access. There are a number of static data files that are fed into the simulation. These largely consist of weather data that is used by a number of the software entities as they perform their functions. These interactions are indicated by green arrows from the files on disk to the software entities that use them.

The last interaction is perhaps the most complex: co-simulation. Utilizing the HELICS co-simulation platform, all the software entities have been written to allow run-time data exchange, enabling the operation of one entity to affect another during execution. This functionality is essential to modeling the scale and complexity of the the system under analysis. The labels on the blue co-simulation data-exchange arrows summarize the data exchanged between the indicated entities.

Market Structure and Interactions

The market structure for the transactive system implemented for the DSO+T was split into two portions: wholesale and retail. The DSO+T study had as a design requirement that the design of the retail market could not require changes to the wholesale market architecture or operation patterns. Thus the market architecture shown in Figure Fig. 3.50 was used as representative of many of the wholesale market structures in the United States.

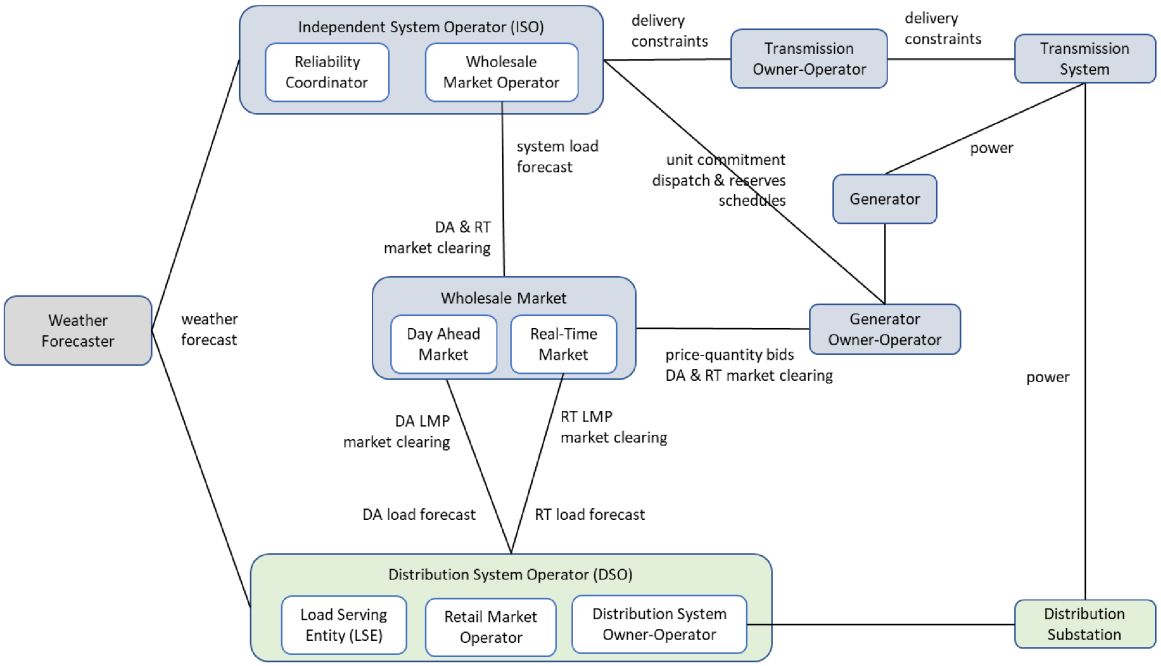


Fig. 3.50: Overview of the wholesale market architecture in the DSO+T study.

The DSO provided load forecasts to the wholesale market (from the retail market it was running) in both the day-ahead and real-time energy markets. The wholesale market treated this as fixed quantity demand bids and with the generator marginal cost bids ran a security constrained unit commitment (day-ahead market only) and/or a security constrained economic dispatch. The former was used to to provide hourly dispatches to the market participants and the later was used to provide five-minute dispatches.

The retail market was designed specifically for the DSO+T study and its structure can be see in Figure Fig. 3.51

The DSO had the responsibility of providing market/load forecast information for all customers in it's jurisdiction and thus had to estimate loads for those not participating the in the transactive system as well as receiving bid information for those participating. Since the communication with the day-ahead market occurred at a specific time and was not communicated as a price-responsive bid curve but a fixed demand quantity, the retail day-ahead market operated in an iterative manner to allow all retail market participants to converge on a day-ahead bid that accounted for their expected flexibility. This iterative process also used weather and solar production forecasts as well as a generic wholesale market marginal cost curve that acted as a wholesale price estimator. After the wholesale markets cleared (day-ahead and real-time), the DSO adjusts these prices to cover their fixed and non-energy marginal costs and communicates these to the market participants. Non-participating customers payed a flat rate that was calculated offline prior to the simulation.

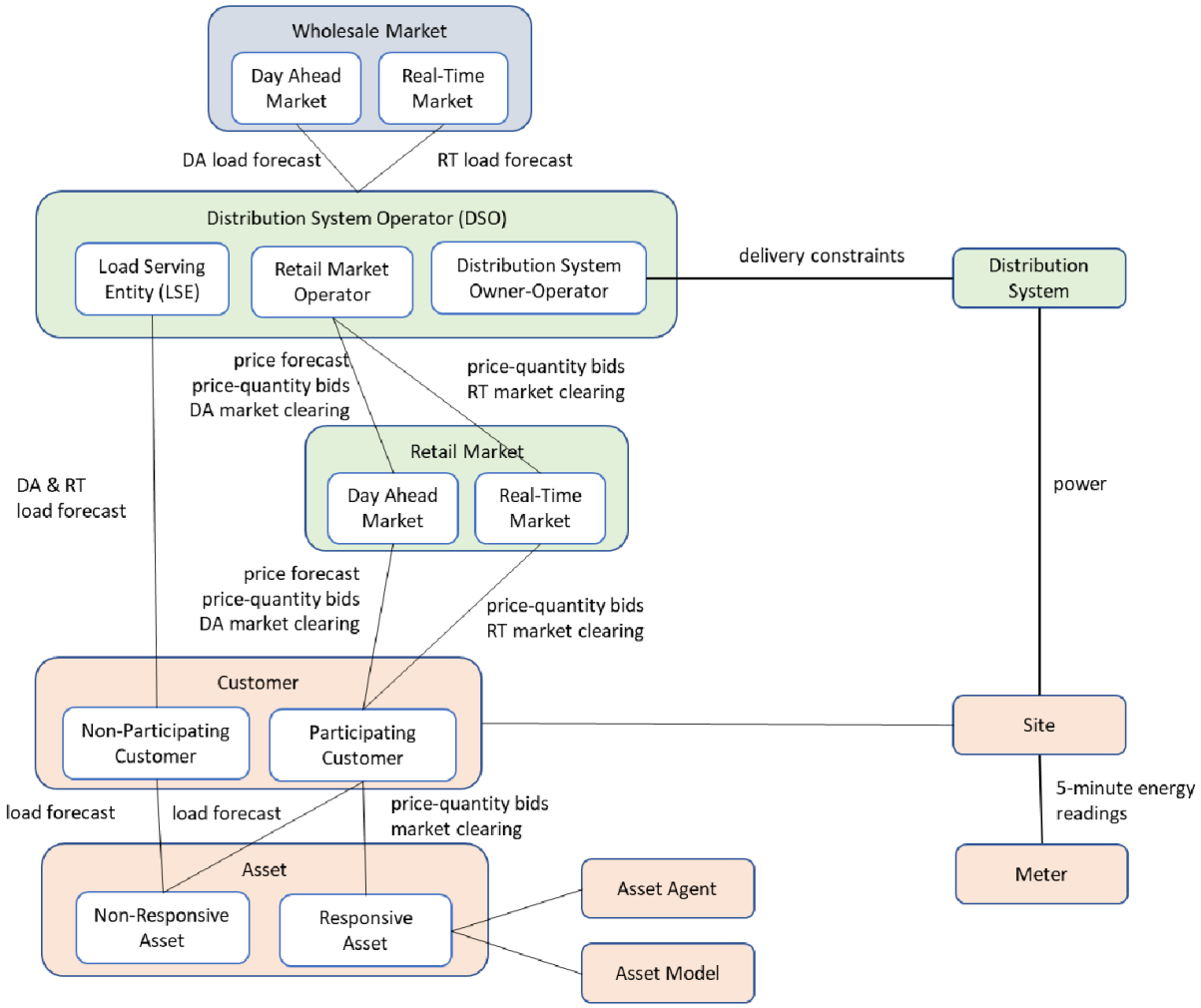


Fig. 3.51: Overview of the retail market architecture in the DSO+T study

Further details on the market and transactive system design can be found in DSO+T: Transactive Energy Coordination Framework DSO+T Study: Volume 3.

Transmission System Model

A simplified 8-bus transmission model was used for the analysis, as shown in Figure Fig. 3.52. A higher-fidelity 200-bus model was used to validate the 8-bus model with similar results. Both models used the same generation fleet where the location of the generators in the 200-bus model were modified to fit the locations available in the 8-bus model. For the high-renewables scenario the existing thermal fleet was maintained while the wind generation capacity was doubled to 32.6 GW, 14.8 GW of utility-scale solar and 21.3 GW of rooftop solar were added (though the rooftop-solar was implemented in the distribution system models). The generation mix for both scenarios are shown in Table 3.7.

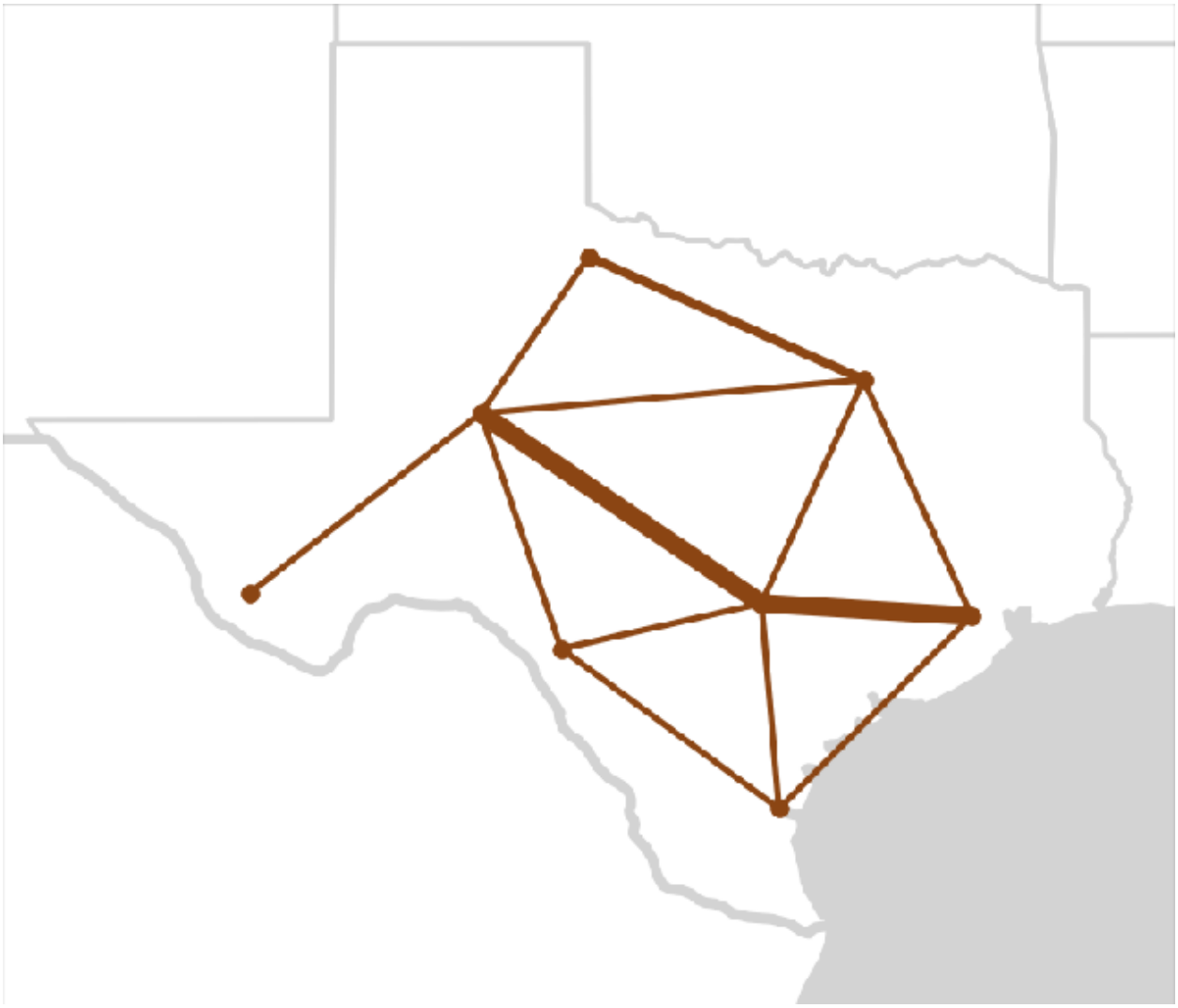


Fig. 3.52: Topology of the simplified 8-bus bulk power system model utilized.

Table 3.7: ERCOT Generation Mix Modeled in DSO+T

Generation Type	Generation Capacity (MW)
Coal	21,900
Natural gas combined cycle	40,100
Natural gas internal combustion engine	1,800
Natural gas steam turbine	13,000
Nuclear	5,100
Wind (MR/HR)	16,300/32,600
Solar (utility scale, HR only)	14,800
Solar (distributed, MR only)	21,300 Total (MR/HR) 98,300/150,600

Further details on the transmission system modeling can be found in Sections 2 and 3 of the DSO+T: Integrated System Simulation DSO+T Study: Volume 2

Distribution System Models

The prototypical feeder models (Github feeder repository, feeder development report) were used as the basis of the distribution system models in DSO+T. Each transmission node with load defined in the transmission system model had one to three of these models combined with a single common substation. These models had their static ZIP loads converted to GridLAB-D house objects to model residential and commercial buildings (less than 30,000 square feet). (Industrial loads were modeled as a constant load directly attached to the transmission system bus.) Extensive literature review was done to help define building and occupant parameters for the models such as building envelope parameters, thermal mass, plug loads and internal gain schedules, HVAC schedules, and water heater types and setpoints.

For the customers participating in the transactive system the HVAC systems, electric water heaters, and EV charging were modeled as the participating loads as these are the highest-power loads. Some of the scenarios also included batteries which, when present, participated in the transactive system. In the high-renewables scenarios some customers had rooftop solar which did not participate as a generator in the transactive system but whose output was considered when estimating the power required by each participant.

Each GridLAB-D model at a given transmission bus used a corresponding TMY3 weather file, resulting in some distribution systems being summer-peaking and some being winter-peaking. The solar production data was calculated using the National Solar Radiation Database.

Figures Fig. 3.53 and Fig. 3.54 show the results for a representative weeks with maximum and minimum load. The gray dashed line shows the actual historic load as measured by ERCOT and the solid black line shows the total simulated load. (The gap between the itemized color load and the black total system load represents system losses.) Though not perfect, the correlation is reasonable and shows that the loads being modeled in the distribution system are generally capturing the behavior of the customer's they represent.

Running DSO+T Example

Due to the scope and scale of the analysis, the DSO+T analysis typically takes several days to simulate a whole month. Setting the simulation duration to a single week will reduce the simulation time to 12-24 hours though the built-in post-processing scripts called by 'postprocess.sh' will not function properly.

Start by downloading supporting data that is not stored in the repository due to its size and static nature. This will add a "data" folder alongside the existing "code" folder from the repository. .. code-block:: sh cd ~/grid/tesp/examples/analysis/dsot/code ./dsotData.sh

Open up "8_system_case_config.json" and confirm/change the following parameters: .. code-block:: sh "StartTime": "2016-08-01 00:00:00" "EndTime": "2016-08-31 00:00:00" "Tmax":<calculate number of seconds in above defined start time> "caseName": <arbitrary name> "dsoPopulationFile": "8-metadata-lean.json"

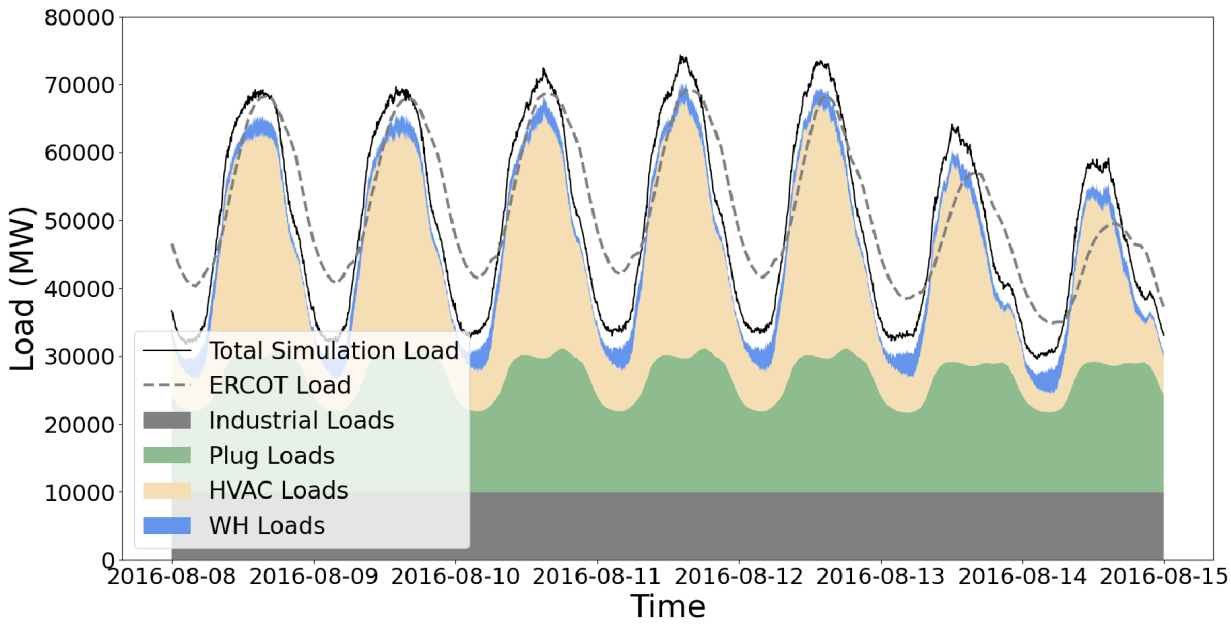


Fig. 3.53: Modeled and historical peak load for ERCOT

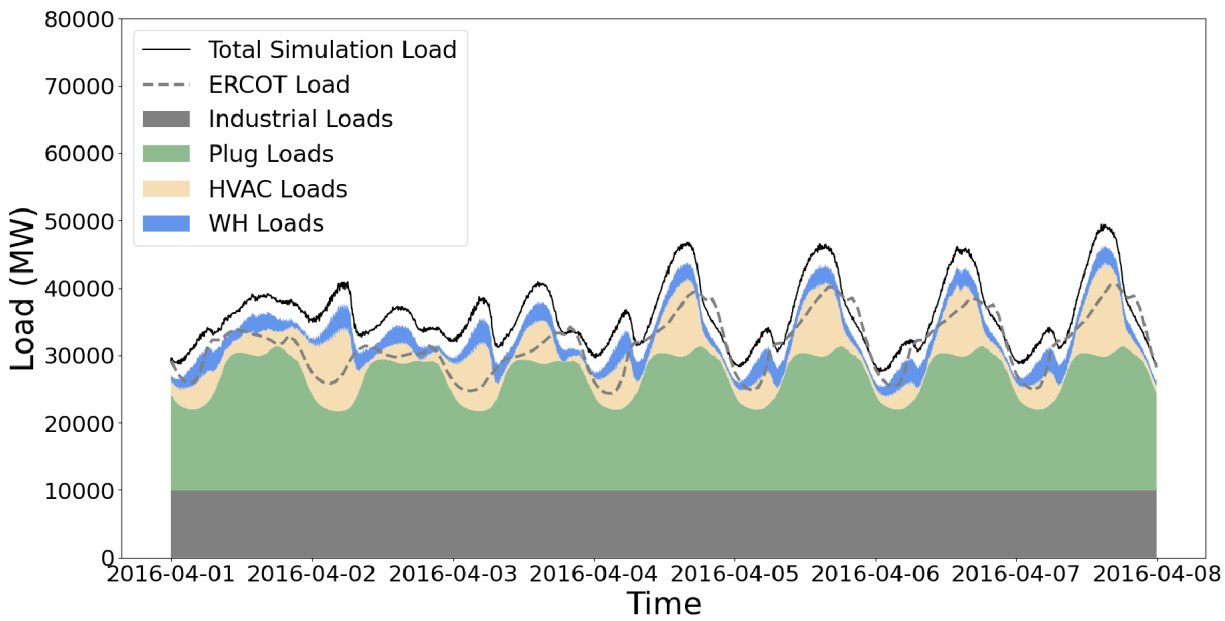


Fig. 3.54: Modeled and historical minimum load for ERCOT

- prepare_case_dsot.py - pre-defined cases are shown; these are the ones used for DSO+T -creates directory in "code"
- postprocess.sh

Results

Below are a sample of the standard plots that are created using the built-in post-processing scripts for this case. These scripts are designed to work on one calendar month of data though the simulation can be configured to run on much shorter periods of time; to process those results custom post-processing scripts will need to be created.

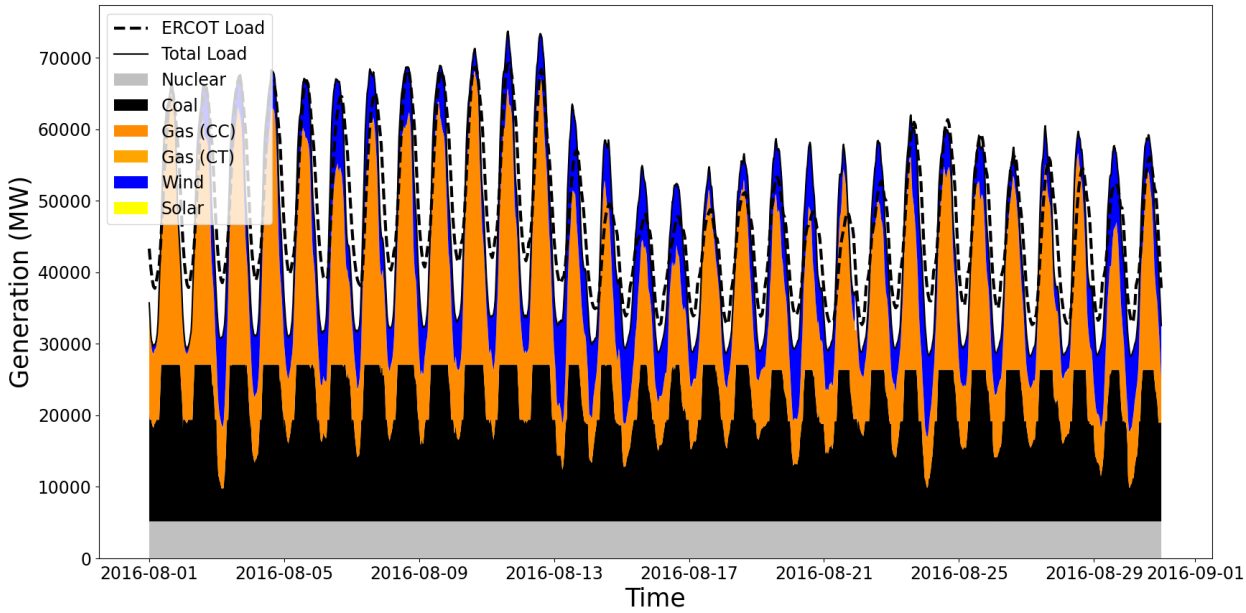


Fig. 3.55: Fuel source for bulk power system generation for the month of August.

3.2.3 Consensus-based Transactive Communities - Example and Documents

This section links to models and design documents for a Consensus-based Transactive Communities use-case that PNNL has been developing as part of the HELICS+ use-cases (4.1) in FY21. Most of this material will be incorporated into the main TESP documentation as the described models and agents are fully integrated with the platform. No technical support can be provided for material referenced from this section, outside of the DSO+T study team.

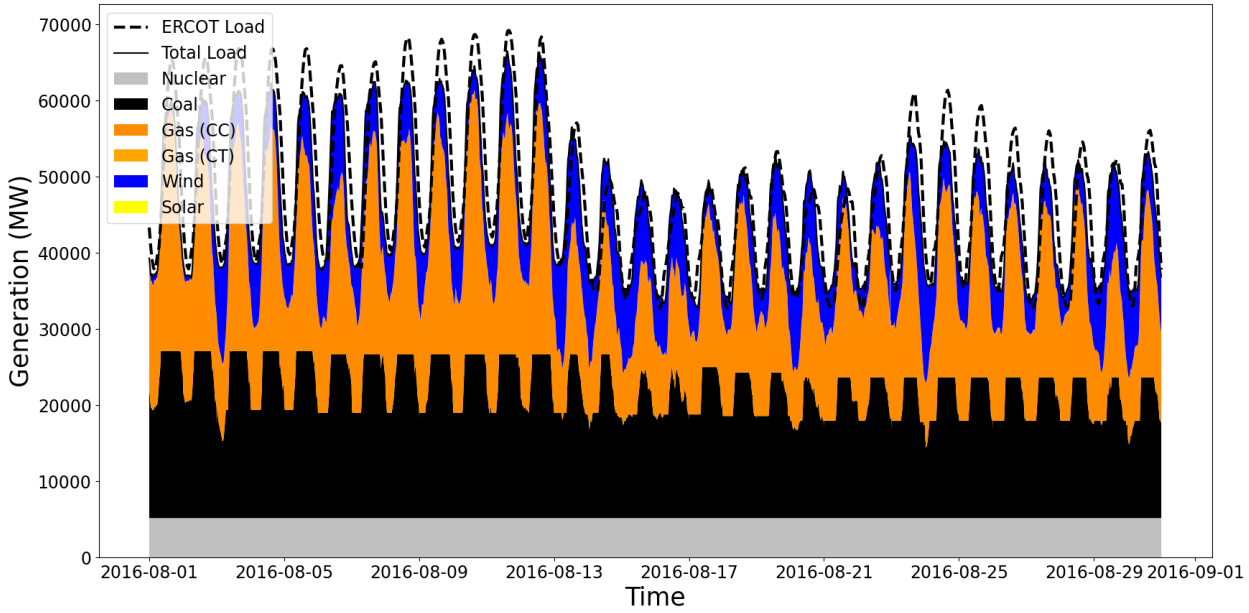


Fig. 3.56: Fuel source for bulk power system generation for the month of August when batteries are installed in the distribution system and participate in a transactive system.

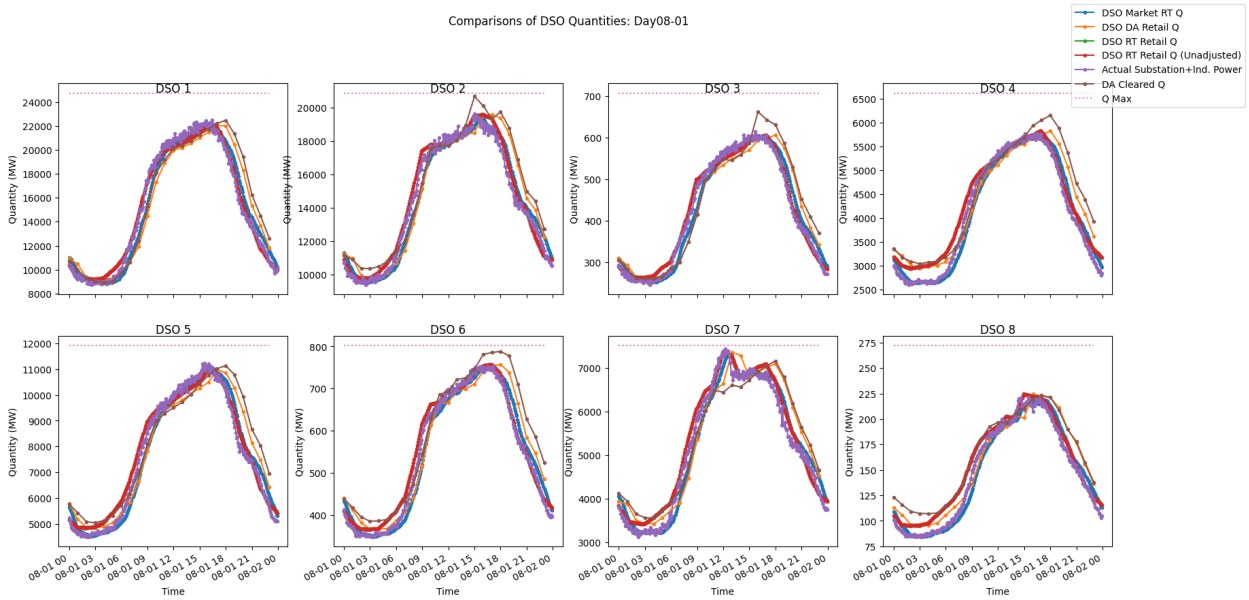


Fig. 3.57: Quantity of energy purchased by each of the eight modeled DSOs in the month of August in the base case.

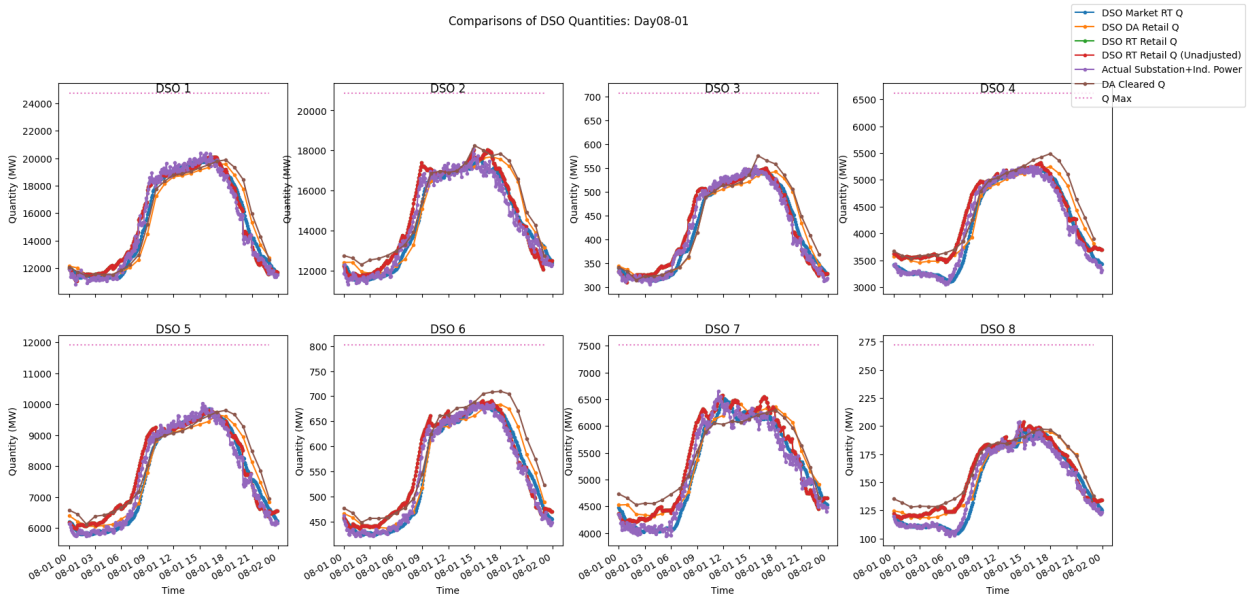


Fig. 3.58: Quantity of energy purchased by each of the eight modeled DSOs in the month of August when batteries are installed in the distribution system and participate in a transactive system.

CHAPTER

FOUR

DEVELOPING AND CUSTOMIZING TESP

4.1 Introduction

Though TESP comes with a collection of capability demonstrations and analysis examples, it exists to smooth the path forward for evaluation of all kinds of transactive energy analysis. We hope users of TESP will not only delve into the aforementioned demonstrations and examples but also seek to customize and expand TESP to their meet their own analysis and research needs. To that end, the goal of this section is to provide the supporting documentation necessary to not only understand TESP as a whole in significant detail but also deconstruct it sufficiently that it is clear to those designing new transactive analysis which pieces can most easily be used.

4.1.1 Standard Third-Party Tools

TESP, as a platform, is largely dependent on third-party simulation tools to perform the kinds of analysis that are common in transactive energy scenarios. The tools summarized on this page are those that have been commonly used in TESP and are considered somewhat "standard" when using TESP. This obviously doesn't preclude the use of other simulation tools but the installation and integration of said tool will fall on the user. When applicable and appropriate, the source code (along with the compiled executable) for these tools is provided with a complete TESP installation and when not, the just the executables are included. By providing access to the source and linking to their repositories, users should be able to not only customize the tools in their installations but also update their code from the repository to receive bug fixes and/or feature updates. The following is a brief description of each of the tools; much more comprehensive documentation for each can be found on their respective websites.

GridLAB-D

GridLAB-D is a power distribution system simulation tool that not only solves three-phase unbalanced power flows (as is common in distribution system tools) but also includes simple thermodynamic models for houses including HVAC systems and water heaters. The inclusion of the multi-domain models allows the tool to model an integrated distribution system (wires and loads) and represent a wider variety of common transactive energy scenarios. GridLAB-D also include models for solar PV installations with inverters, automated voltage management equipment (voltage regulators and switched capacitors), and diesel generators. TESP uses GridLAB-D to model distribution system physics and customer load behavior.

PYPOWER

PYPOWER is a Python re-implementation of MATPOWER in Python using common Python libraries to perform the mathematical heavy lifting. TESP uses PYPOWER to solve the real-time energy market dispatch (through an optimal power flow) and the transmission system physics (through traditional power flows).

PSST

Power system solver that provides an alternative formulation and implementation for solving day-ahead security-constrained unit commitment (SCUC) and real-time security-constrained economic dispatch (SCED) problems.

EnergyPlus

EnergyPlus is a building system simulator that is typically used to model large (at least relative to residential buildings), multi-zone commercial buildings. These models include representations of both the physical components/structures of the buildings (*e.g.* walls, windows, roofs) but also heating and cooling equipment as well as their associated controllers. TESP uses EnergyPlus to model commercial structures and associate them with particular points in the GridLAB-D model.

ns-3

ns-3 is a discrete-event communication system simulator that TESP uses to model communication system effects between the various agents in a transactive system. ns-3 has built-in models to present common protocol stacks (TCP/IP, UDP/IP), wireless protocols such as Wifi (802.11) and LTE networks, as well as various routing protocols (*e.g.* OLSR, AODV).

HELICS

HELICS is a co-simulation platform that is used to integrate all of the above tools along with the custom agents created by the TESP team and distributed with TESP. HELICS allows the passing of physical values or abstract messages between the TESP simulation tools during run-time allowing each simulation tool to affect the others. For example, PYPOWER produces a substation voltage for GridLAB-D and GridLAB-D, using that voltage, produces a load for PYPOWER.

Ipopt

Quoting from it's website, '"Ipopt (Interior Point Optimizer, pronounced "Eye-Pea-Opt") is an open source software package for large-scale nonlinear optimization." Ipopt can be used by any other software in TESP that performs optimization problems, most typically in solving multi-period optimization problems such as in solving the day-ahead energy market or devices creating day-ahead bids for participating in said markets. Ipopt is built with Ampl Solver Library (ASL) and and MUMPS support.

Python Packages

There are many Python packages used but a few of the major ones not already listed deserve mention:

- Matplotlib - data visualization library used for presenting results out of TESP
- NumPy - data management library used for structuring results data for post-processing
- pandas - data management library used for structuring results data for post-processing
- HDF5 - Database-like data format used for storing results from some simulation tools and used to read in said data for post-processing
- seaborn - data visualization library used for presenting results out of TESP
- Pyomo - optimization modeling language used to formulate some of the multi-period optimizations common in TESP
- Networkx - graph modeling package used to analyze some of the relational graphs and/or models of the power and communication network in TESP

CHAPTER

FIVE

REFERENCES

References for TESP

5.1 Design Reference

5.1.1 Messages between Simulators and Agents

TESP simulators exchange the sets of messages shown in Fig. 5.1 and Fig. 5.2.

These messages route through HELICS in a format like "topic/keyword=value". In Fig. 5.3, the "id" would refer to a specific feeder, house, market, or building, and it would be the message topic. Once published via HELICS, any other HELICS simulator can access the value by subscription. For example, PYPOWER publishes two values, the locational marginal price (LMP) at a substation bus and the positive sequence three-phase voltage at the bus. GridLAB-D subscribes to the voltage, using it to update the power flow solution. The double-auction for that substation subscribes to the LMP, using it to represent a seller in the next market clearing interval. In turn, GridLAB-D publishes a distribution load value at the substation following each significantly different power flow solution; PYPOWER subscribes to that value for its next optimal power flow solution.

EnergyPlus publishes three phase power values after each of its solutions (currently on five-minute intervals). These are all numerically equal, at one third of the total building power that includes lights, office equipment, refrigeration and HVAC loads. GridLAB-D subscribes in order to update its power flow model at the point of interconnection for the building, which is typically at a 480-V or 208-V three-phase transformer. EnergyPlus also subscribes to the double-auction market's published clearing price, using that value for a real-time price (RTP) response of its HVAC load.

Message flows involving the thermostat controller, at the center of Fig. 5.3, are a little more involved. From the associated house within GridLAB-D, it subscribes to the air temperature, HVAC power state, and the HVAC power if turned on. The controller uses this information to help formulate a bid for electric power at the next market clearing, primarily the price and quantity. Note that each market clearing interval will have its own market id, and that re-bidding may be allowed until that particular market id closes. When bidding closes for a market interval, the double-auction market will settle all bids and publish several values, primarily the clearing price. The house thermostat controllers use that clearing price subscription, compared to their bid price, to adjust the HVAC thermostat setpoint. As noted above, the EnergyPlus building also uses the clearing price to determine how much to adjust its thermostat setting. Fig. 5.3 shows several other keyword values published by the double-auction market and thermostat controllers; these are mainly used to define "ramps" for the controller bidding strategies. See the GridLAB-D documentation, or TESP design documentation, for more details.

These message schemas are limited to the minimum necessary to operate Version 1, and it's expected that the schema will expand as new TEAgents are added. Beyond that, note that any of the simulators may subscribe to any values that it "knows about", i.e., there are no security and access control emulations. This may be a layer outside the scope of TESP. However, there is also no provision for enforcement of bid compliance, i.e. perfect compliance is built into the code. That's clearly not a realistic assumption, and is within the scope for future versions as described in Section 3.

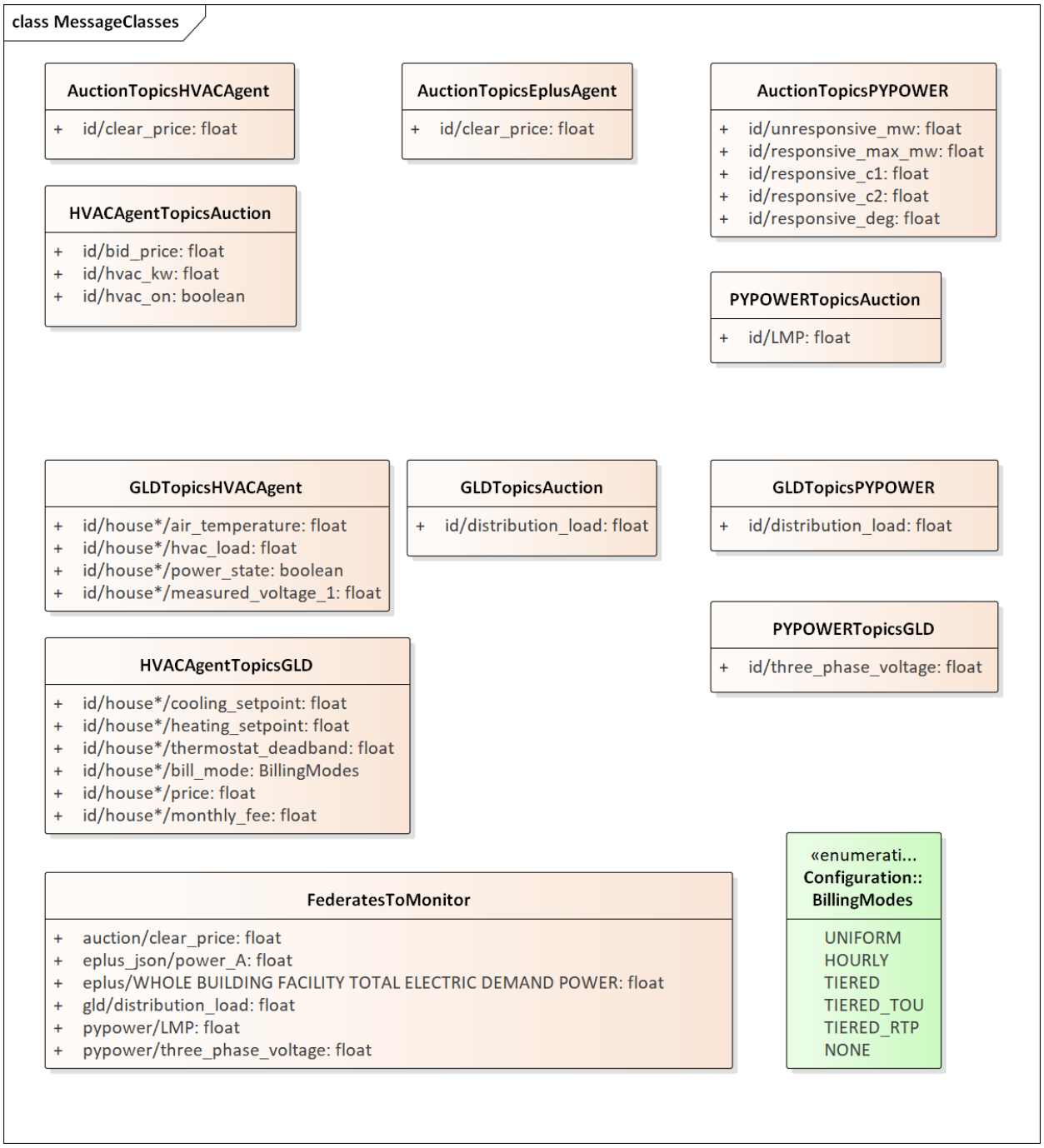


Fig. 5.1: Message Schemas for GridLAB-D, PYPOWER and residential agents

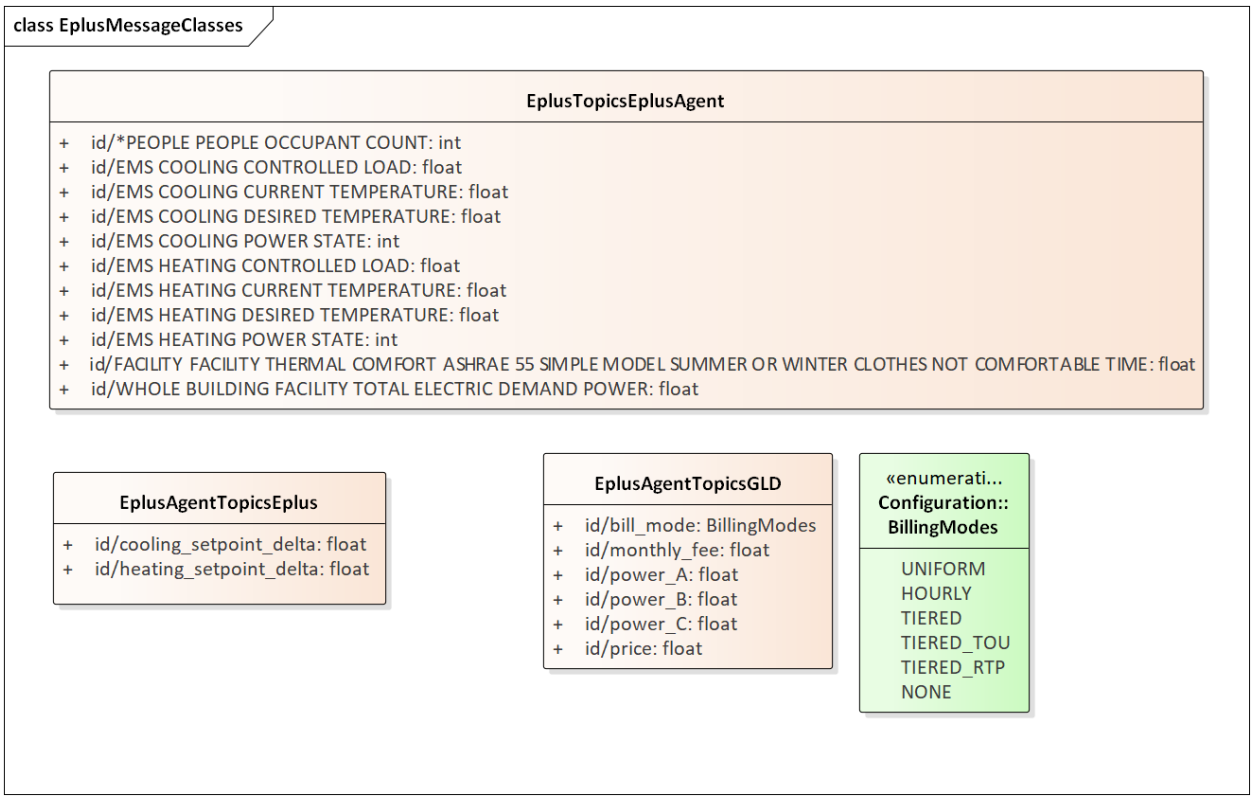


Fig. 5.2: Message Schemas for EnergyPlus and large-building agents

5.1.2 TESP for Agent Developers

The left-hand portion of Fig. 5.5 shows the main simulators running in TESP and communicating over HELICS. For the DSO+T study, PYPOWER will be upgraded to AMES and EnergyPlus will be upgraded to a Modelica-based large-building simulator. The large-building agent will also be updated. The current large-building agent is written in C++. It's functionality is to write metrics from EnergyPlus, and also to adjust a thermostat slider for the building. However, it does not formulate bids for the building. The right-hand portion of Fig. 5.5 shows the other transactive agents implemented in Python. These communicate directly via Python function calls, i.e., not over HELICS. There are several new agents to implement for the DSO+T study, and this process will require four main tasks:

1 - Define the message schema for information exchange with GridLAB-D, AMES or other HELICS federates. The SubstationAgent will actually manage the HELICS messages, i.e., the agent developer will not be writing HELICS interface code.

2 - Design the agent initialization from metadata available in the GridLAB-D or other dictionaries, i.e., from the dictionary JSON files.

3 - Design the metadata for any intermediate metrics that the agent should write (to JSON files) at each time step.

4 - Design and implement the agent code as a Python class within tesp_support. The SubstationAgent will instantiate this agent class at runtime, and call the class as needed in a time step.

Fig. 5.6 the sequence of interactions between GridLAB-D, the SubstationAgent (encapsulating HVAC controllers and a double-auction market) and PYPOWER. The message hops over HELICS each consume one time step. The essential messages for market clearing are highlighted in red. Therefore, it takes 3 HELICS time steps to complete a market clearing, from the collection of house air temperatures to the adjustment of thermostat setpoints. Without the encapsulating SubstationAgent, two additional HELICS messages would be needed. The first would occur between the self-messages AgentBids and Aggregate, routed between separate HVACController and Auction swimlanes. The

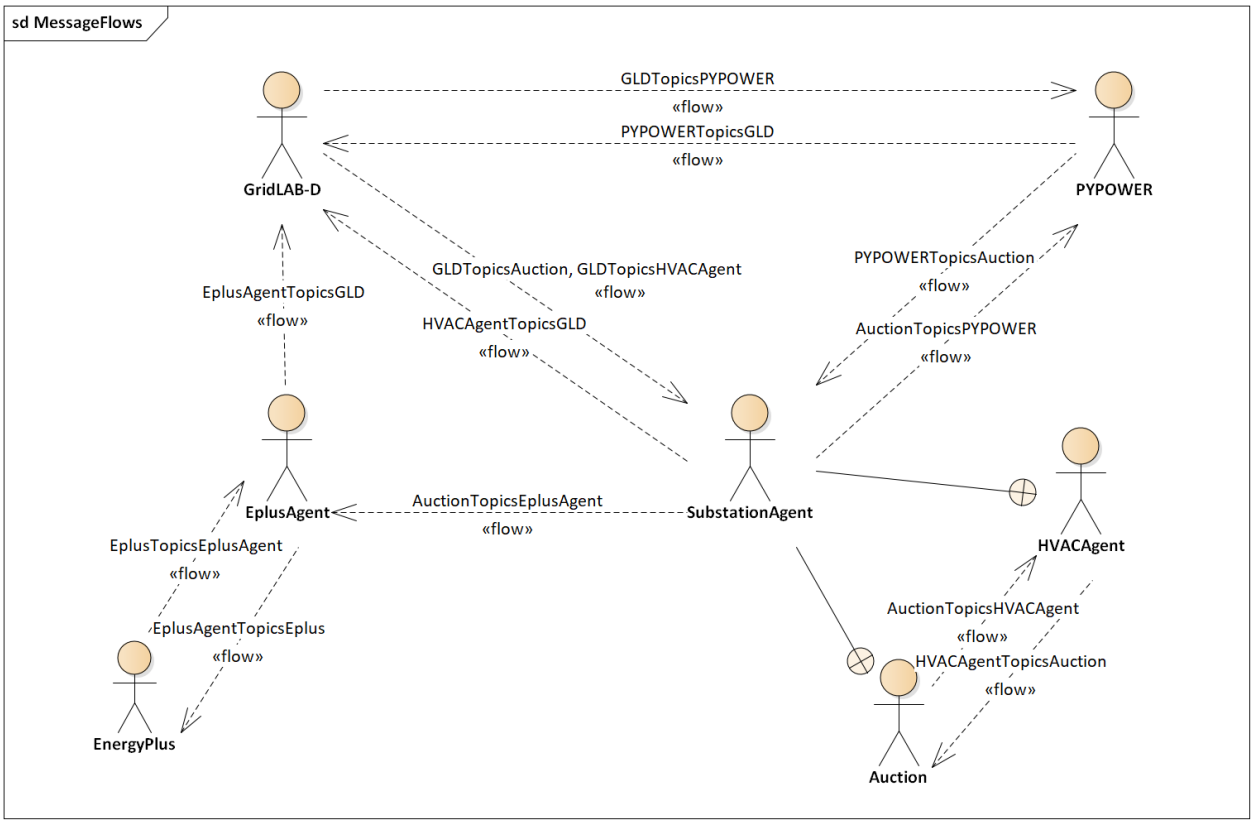


Fig. 5.3: Message Flows for Simulators and Transactive Agents

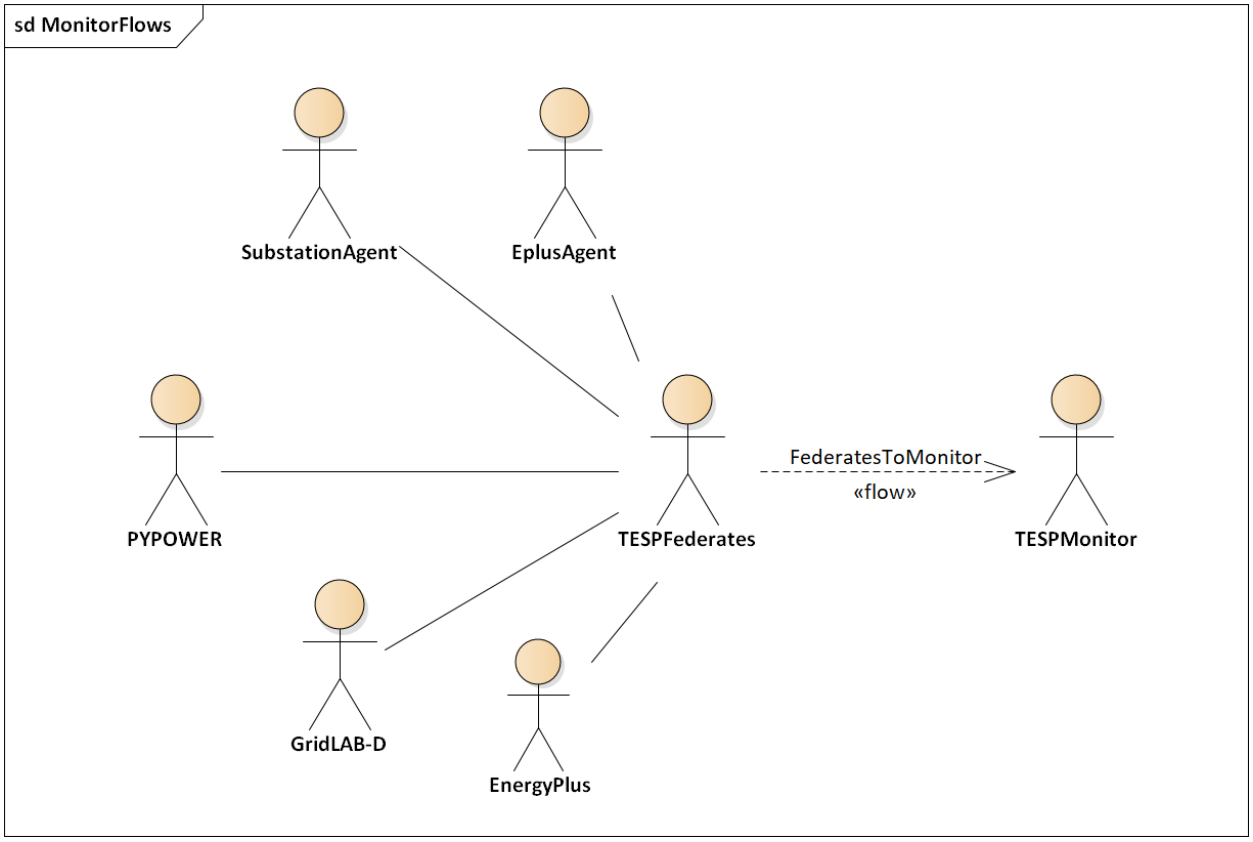


Fig. 5.4: Message Flows for Solution Monitoring

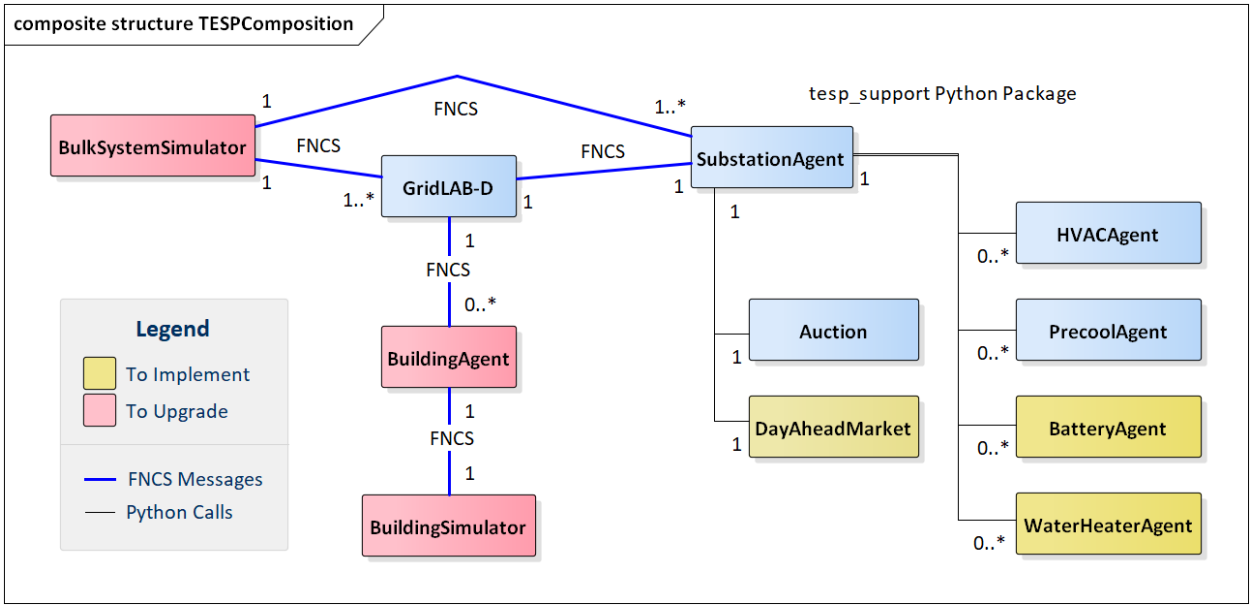


Fig. 5.5: Composition of Federates in a Running TESP Simulation

second would occur between ClearMarket and AdjustSetpoints messages, also routed between separate Auction and HVACController swimlanes. This architecture would produce additional HELICS message traffic, and also increase the market clearing latency from 3 HELICS time steps to 5 HELICS time steps.

Before and after each market clearing, GridLAB-D and PYPOWER will typically exchange substation load and bus voltage values several times, for each power flow (PF) solution. These HELICS messages are indicated in black; they represent much less traffic than the market clearing messages.

Some typical default time steps are:

1 - 5 seconds, for HELICS, leading to a market clearing latency of 15 seconds

2 - 15 seconds, for GridLAB-D and PYPOWER's regular power flow (PF)

3 - 300 seconds, for spot-market clearing, PYPOWER's optimal power flow (OPF), EnergyPlus (not shown in Fig. 5.6) and the metrics aggregation.

5.1.3 Output Metrics to Support Evaluation

TESP will produce various outputs that support comparative evaluation of different scenarios. Many of these outputs are non-monetary, so a user will have to apply different weighting and aggregation methods to complete the evaluations. This is done in the Evaluation Script, which is written in Python. These TESP outputs all come from the Operational Model, or from the Growth Model applied to the Operational Model. For efficiency, each simulator writes intermediate metrics to Javascript Object Notation (JSON) files during the simulation, as shown in Figure 5. For example, if GridLAB-D simulates a three-phase commercial load at 10-second time steps, the voltage metrics output would only include the minimum, maximum, mean and median voltage over all three phases, and over a metrics aggregation interval of 5 to 60 minutes. This saves considerable disk space and processing time over the handling of multiple CSV files. Python, and other languages, have library functions optimized to quickly load JSON files.

To support these intermediate metrics, two new classes were added to the "tape" module of GridLAB-D, as shown in Fig. 5.8. The volume and variety of metrics generated from GridLAB-D is currently the highest among simulators within TESP, so it was especially important here to provide outputs that take less time and space than CSV files. Most of the outputs come from billing meters, either single-phase triplex meters that serve houses, or three-phase meters that serve commercial loads. The power, voltage and billing revenue outputs are linked to these meters, of which there may be several thousand on a feeder. Houses, which always connect to triplex meters, provide the air temperature and setpoint deviation outputs for evaluating occupant comfort. Inverters, which always connect to meters, provide real and reactive power flow outputs for connected solar panels, battery storage, and future DER like vehicle chargers. Note that inverters may be separately metered from a house or commercial building, or combined on the same meter as in net metering. Feeder-level metrics, primarily the real and reactive losses, are also collected by a fourth class that iterates over all transformers and lines in the model; this substation-level class has just one instance not shown in Fig. 5.8. An hourly metrics output interval is shown, but this is adjustable.

The initial GridLAB-D metrics are detailed in five UML diagrams, so we begin the UML metric descriptions with PYPOWER, which is much simpler. During each simulation, PYPOWER will produce two JSON files, one for all of the generators and another for all of the HELICS interface buses to GridLAB-D. A third JSON file, called the dictionary, is produced before the simulation starts from the PYPOWER case input file. The dictionary serves as an aid to post-processing. Fig. 5.9 shows the schema for all three PYPOWER metrics files.

The PYPOWER dictionary (top of Fig. 5.9) includes the system MVA base (typically 100) and GridLAB-D feeder amplification factor. The amplification factor is used to scale up the load from one simulated GridLAB-D feeder to represent many similar feeders connected to the same PYPOWER bus. Each generator has a bus number (more than one generator can be at a bus), power rating, cost function $f(P) = c_0 + c_1 P + c_2 P^2$, startup cost, shutdown cost, and other descriptive information. Each DSOBus has nominal P and Q that PYPOWER can vary outside of GridLAB-D, plus the name of a GridLAB-D substation that provides additional load at the bus. In total, the PYPOWER dictionary contains four JSON objects; the *ampFactor*, the *baseMVA*, a dictionary (map) of Generators keyed on the generator id, and a dictionary (map) of DSOBuses keyed on the bus id. In PYPOWER, all id values are integers, but the other simulators use string ids.

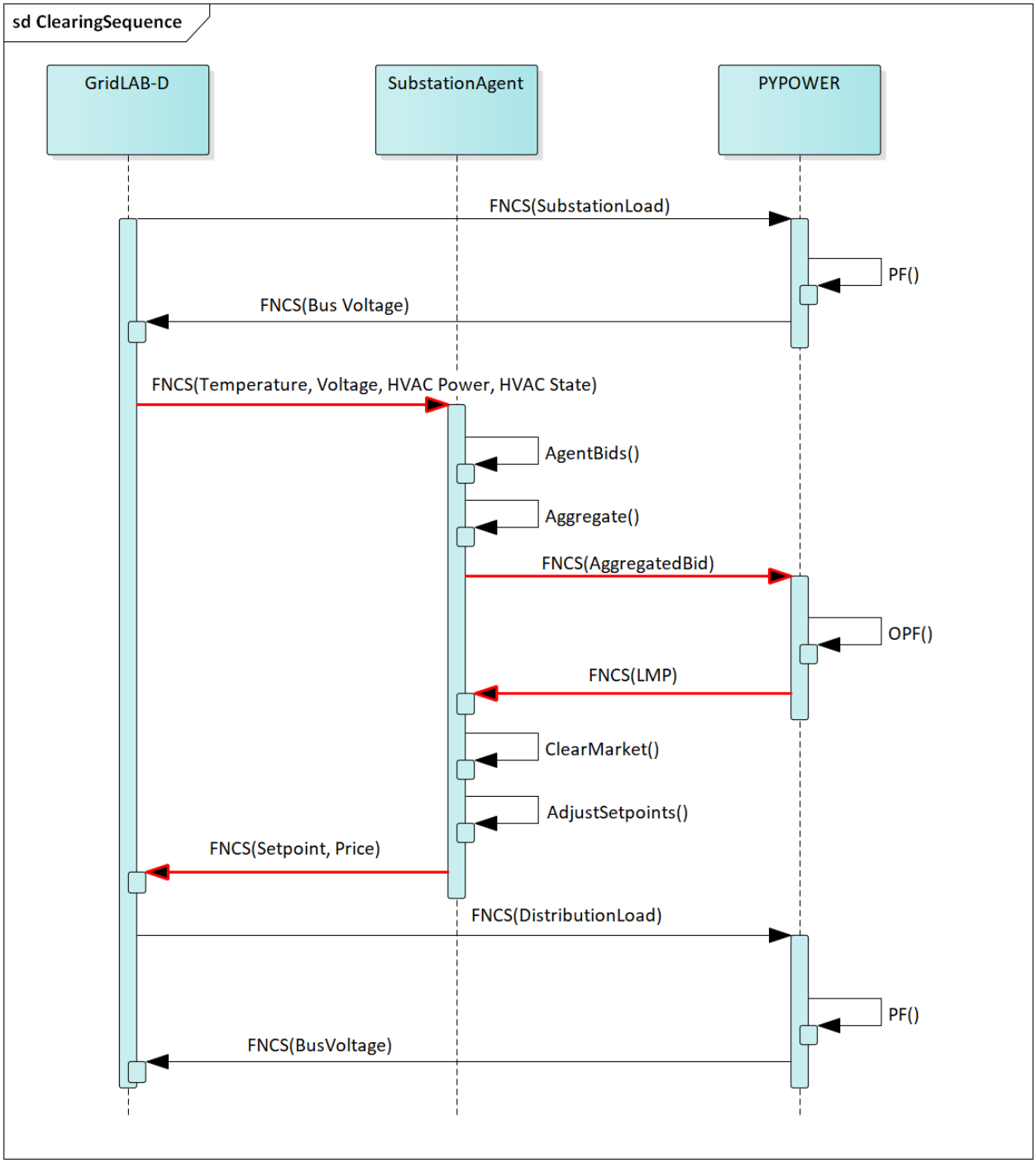


Fig. 5.6: HELICS Message Hops around Market Clearing Time

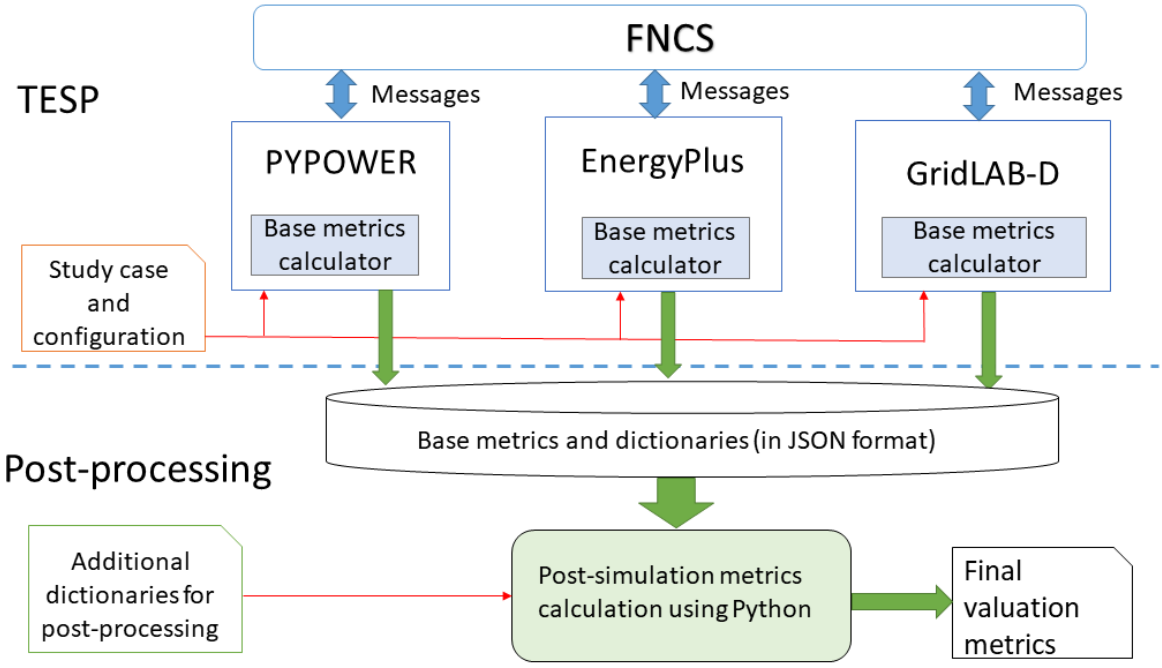


Fig. 5.7: Partitioning the valuation metrics between simulation and post-processing

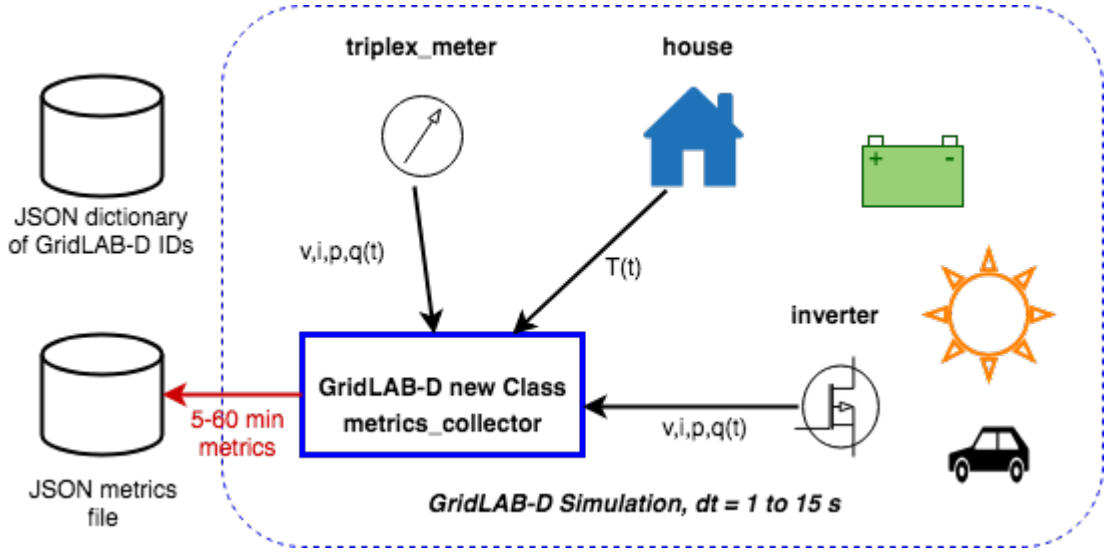


Fig. 5.8: New metrics collection classes for GridLAB-D

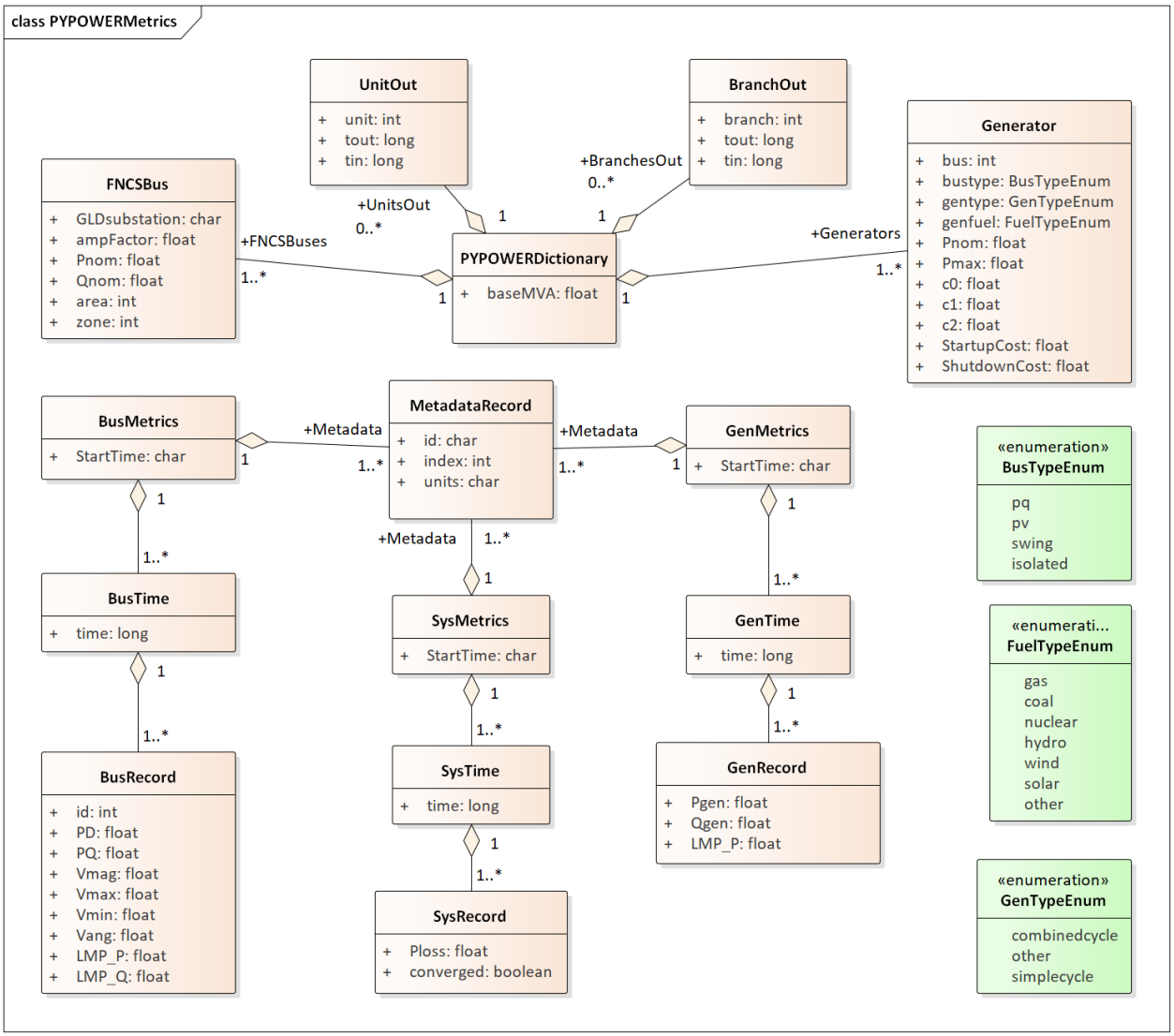


Fig. 5.9: PYPOWER dictionary with generator and DSO bus metrics

The GenMetrics file (center of Fig. 5.9) includes the simulation starting date, time and time zone as *StartTime*, which should be the same in all metrics output files from that simulation. It also contains a dictionary (map) of three MetadataRecords, which define the array index and units for each of the three generator metric output values. These are the real power *LMP*, along with the actual real and reactive power outputs, *Pgen* and *Qgen*. At each time for metrics output, a GenTime dictionary (map) object will be written with key equal to the time in seconds from the simulation *StartTime*, and the value being a dictionary (map) of GenRecords.

The GenRecord keys are generator numbers, which will match the dictionary. The GenRecord values are arrays of three indexed output values, with indices and units matching the Metadata. This structure minimizes nesting in the JSON file, and facilitates quick loading in a Python post-processor program. Valuation may require the use of both metrics and the dictionary. For example, suppose we need the profit earned by a generator at a time 300 seconds after the simulation starting time. The revenue comes from the metrics as *LMP_P * Pgen*. In order to find the cost, one would start with cost function coefficients obtained from the dictionary for that generator, and substitute *Pgen* into that cost function. In addition, the post processing script should add startup and shutdown costs based on *Pgen* transitions between zero and non-zero values; PYPOWER itself does not handle startup and shutdown costs. Furthermore, aggregating across generators and times would have to be done in post-processing, using built-in functions from Python's NumPy package. The repository includes an example of how to do this.

Turning to more complicated GridLAB-D metrics, Fig. 5.10 provides the dictionary. At the top level, it includes the substation transformer size and the PYPOWER substation name for HELICS connection. There are four dictionaries (maps) of component types, namely houses, inverters, billing meters and feeders. While real substations often have more than one feeder, in this model only one feeder dictionary will exist, comprising all GridLAB-D components in that model. The reason is that feeders are actually distinguished by their different circuit breakers or reclosers at the feeder head, and GridLAB-D does not currently associate components to switches that way. In other words, there is one feeder and one substation per GridLAB-D file in this version of TESP. When this restriction is lifted in a future version, attributes like *feeder_id*, *house_count* and *inverter_count* will become helpful. At present, all *feeder_id* attributes will have the same value, while *house_count* and *inverter_count* will simply be the length of their corresponding JSON dictionary objects. Fig. 5.10 shows that a BillingMeter must have at least one House or Inverter with no upper limit, otherwise it would not appear in the dictionary. The *wh_gallons* attribute can be used to flag a thermostat-controlled electric waterheater, but these are not yet treated as responsive loads in Version 1. Other attributes like the inverter's *rated_W* and the house's *sqft* could be useful in weighting some of the metric outputs.

Fig. 5.11 shows the structure of substation metrics output from GridLAB-D, consisting of real power and energy, reactive power and energy, and losses from all distribution components in that model. As with PYPOWER metrics files, the substation metrics JSON file contains the *StartTime* of the simulation, Metadata with array index and units for each metric value, and a dictionary (map) of time records, keyed on the simulation time in seconds from *StartTime*. Each time record contains a dictionary (map) of SubstationRecords, each of which contains an array of 18 values. This structure, with minimal nesting of JSON objects, was designed to facilitate fast loading and navigation of arrays in Python. The TESP code repository includes examples of working with metrics output in Python. Fig. 5.12 and Fig. 5.13 show how capacitor switching and regulator tap changing counts are captured as metrics.

Fig. 5.14 shows the structure of billing meter metrics, which is very similar to that of substation metrics, except that each array contains 30 values. The billing meter metrics aggregate real and reactive power for any houses and inverters connected to the meter, with several voltage magnitude and unbalance metrics. The interval bill is also included, based on metered consumption and the tariff that was input to GridLAB-D. In some cases, revenues may be recalculated in post-processing to explore different tariff designs. It's also possible to re-calculate the billing determinants from metrics that have been defined.

The Range A and Range B metrics in Fig. 5.14 refer to ANSI C84.1 [3]. For service voltages less than 600 V, Range A is +/- 5% of nominal voltage for normal operation. Range B is -8.33% to +5.83% of nominal voltage for limited-extent operation. Voltage unbalance is defined as the maximum deviation from average voltage, divided by average voltage, among all phases present. For three-phase meters, the unbalance is based on line-to-line voltages, because that is how motor voltage unbalance is evaluated. For triplex meters, unbalance is based on line-to-neutral voltages, because there is only one line-to-line voltage. In Fig. 5.14, *voltage_* refers to the line-to-neutral voltage, while *voltage12_* refers to the line-to-line voltage. The *below_10_percent* voltage duration and count metrics indicate when the billing meter has no voltage. That information would be used to calculate reliability indices in post-processing, with flexible weighting

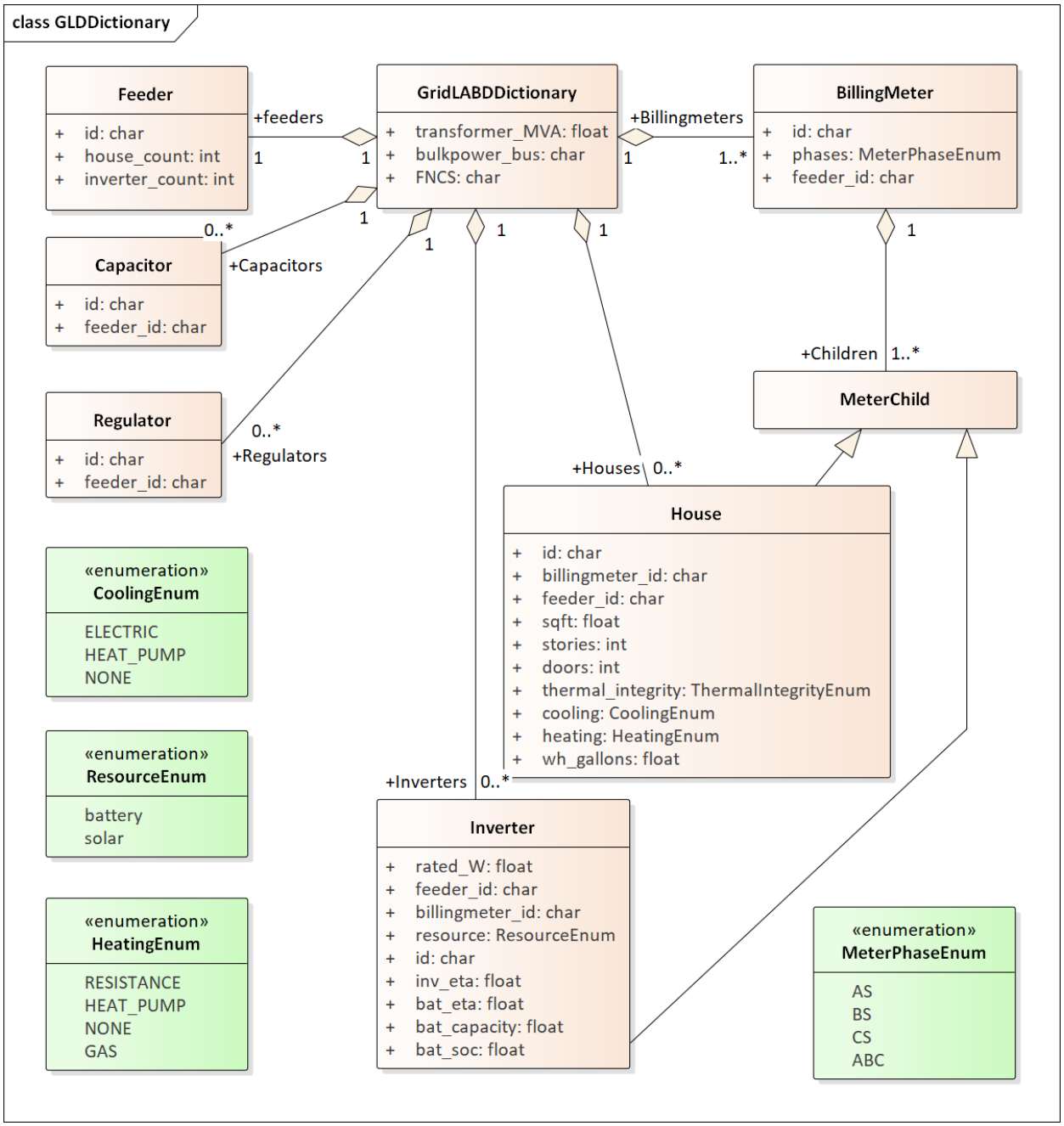


Fig. 5.10: GridLAB-D dictionary

and aggregation options by customer, owner, circuit, etc. These include the System Average Interruption Frequency Index (SAIFI) and System Average Interruption Duration Index (SAIDI) [17, 18]. This voltage-based approach to reliability indices works whether the outage resulted from a distribution, transmission, or bulk generation event. The voltage-based metrics also support Momentary Average Interruption Frequency Index (MAIFI) for shorter duration outages.

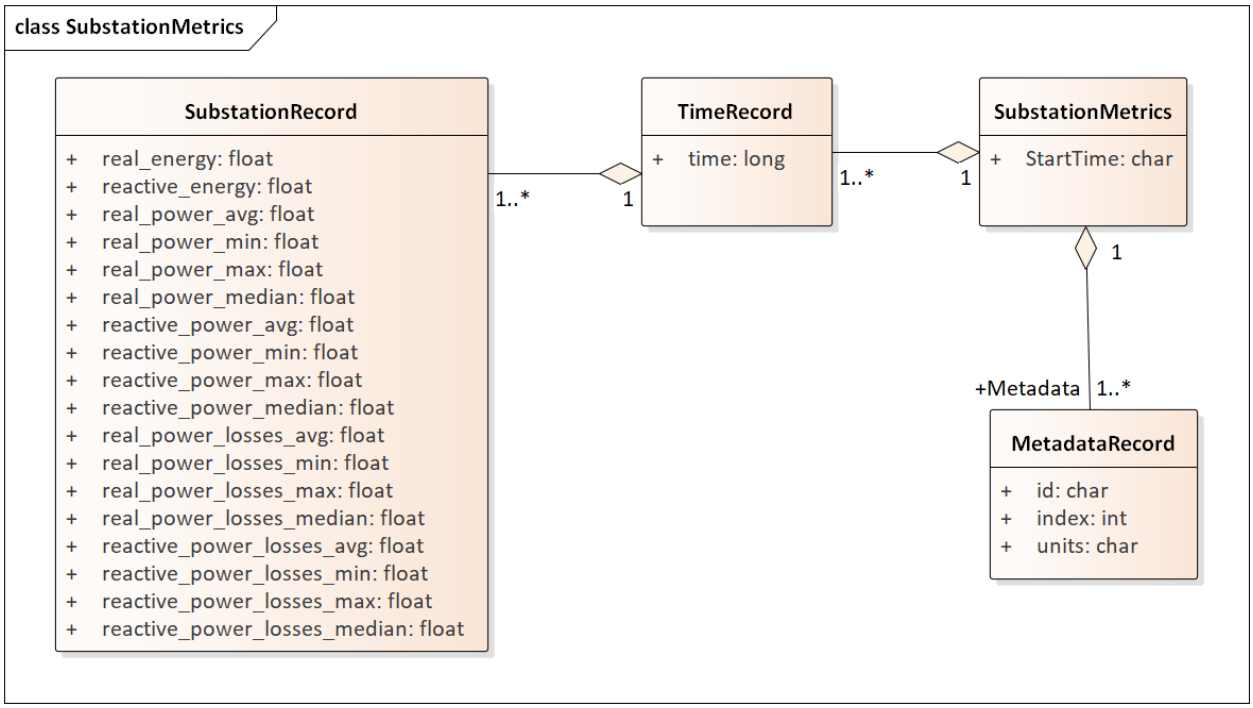


Fig. 5.11: GridLAB-D substation metrics

The house metric JSON file structure is shown in Fig. 5.15, following the same structure as the other GridLAB-D metrics files, with 18 values in each array. These relate to the breakdown of total house load into HVAC and waterheater components, which are both thermostat controlled. The house air temperature, and its deviation from the thermostat setpoint, are also included. Note that the house bill would be included in billing meter metrics, not the house metrics. Inverter metrics in Fig. 5.16 include 8 real and reactive power values in the array, so the connected resource outputs can be disaggregated from the billing meter outputs, which always net the connected houses and inverters. In Version 1, the inverters will be net metered, or have their own meter, but they don't have transactive agents yet.

Sample of resulting JSON:

```
{
"Metadata": {
    "reactive_power_avg": {
        "index": 5,
        "units": "VAR"
    },
    "reactive_power_max": {
        "index": 4,
        "units": "VAR"
    },
    "reactive_power_min": {
        "index": 3,
        "units": "VAR"
```

(continues on next page)

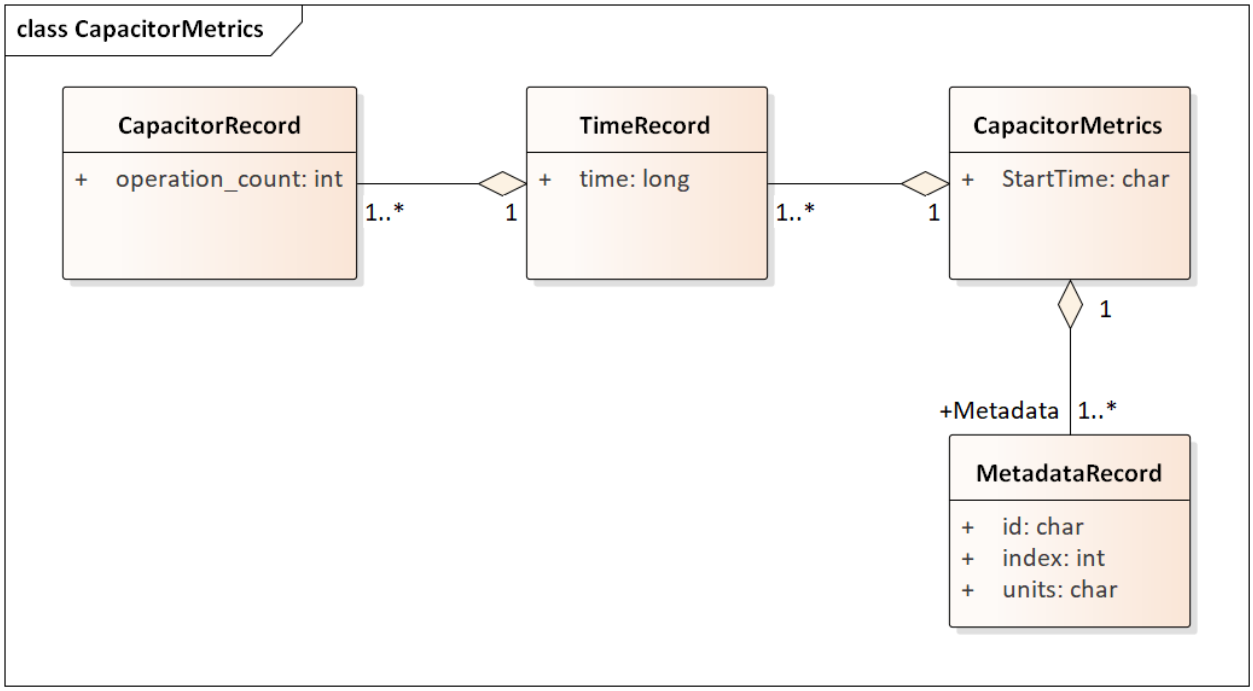


Fig. 5.12: GridLAB-D capacitor switching metrics

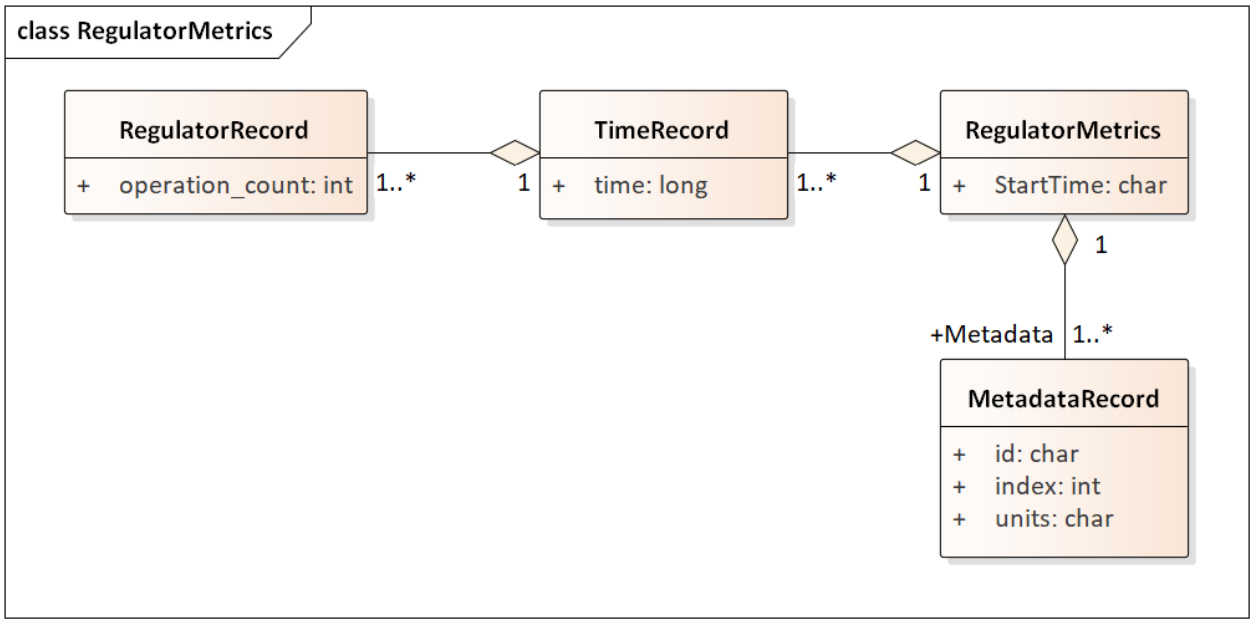


Fig. 5.13: GridLAB-D regulator tap changing metrics

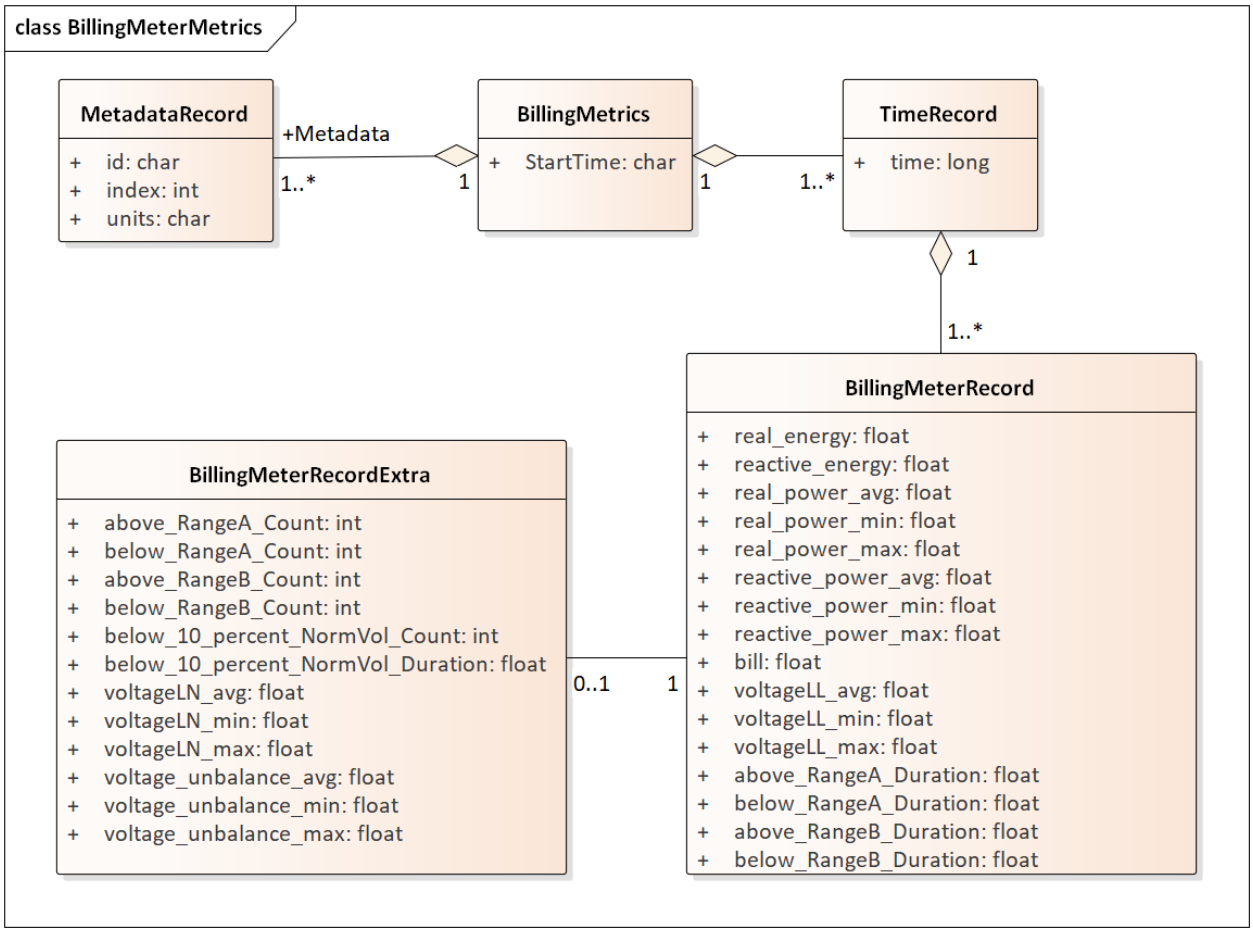


Fig. 5.14: GridLAB-D billing meter metrics

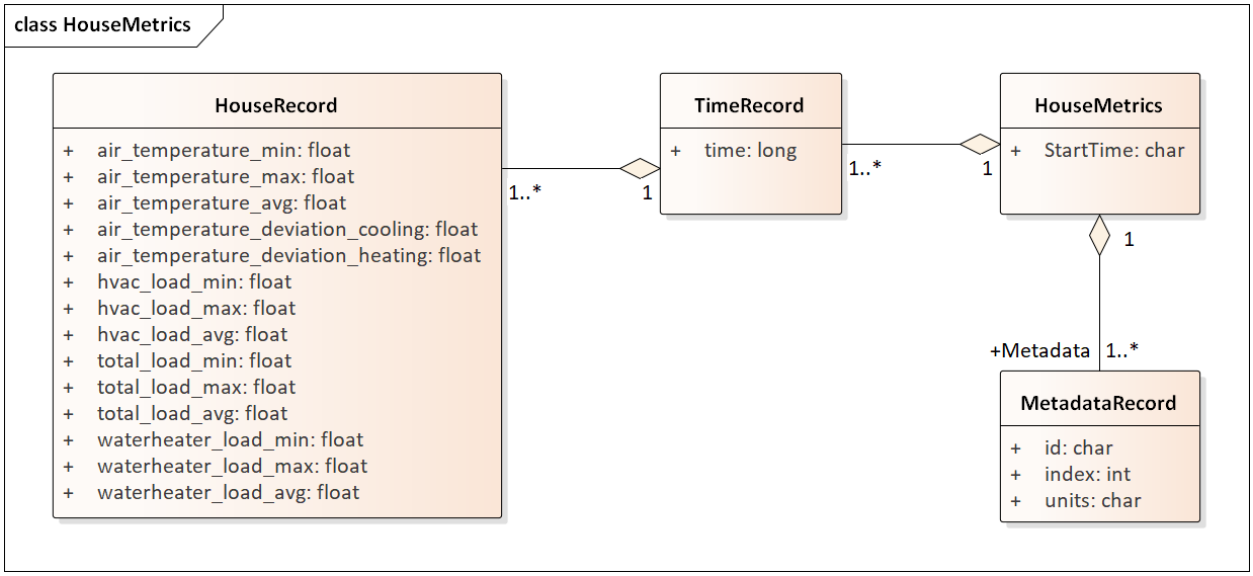


Fig. 5.15: GridLAB-D house metrics

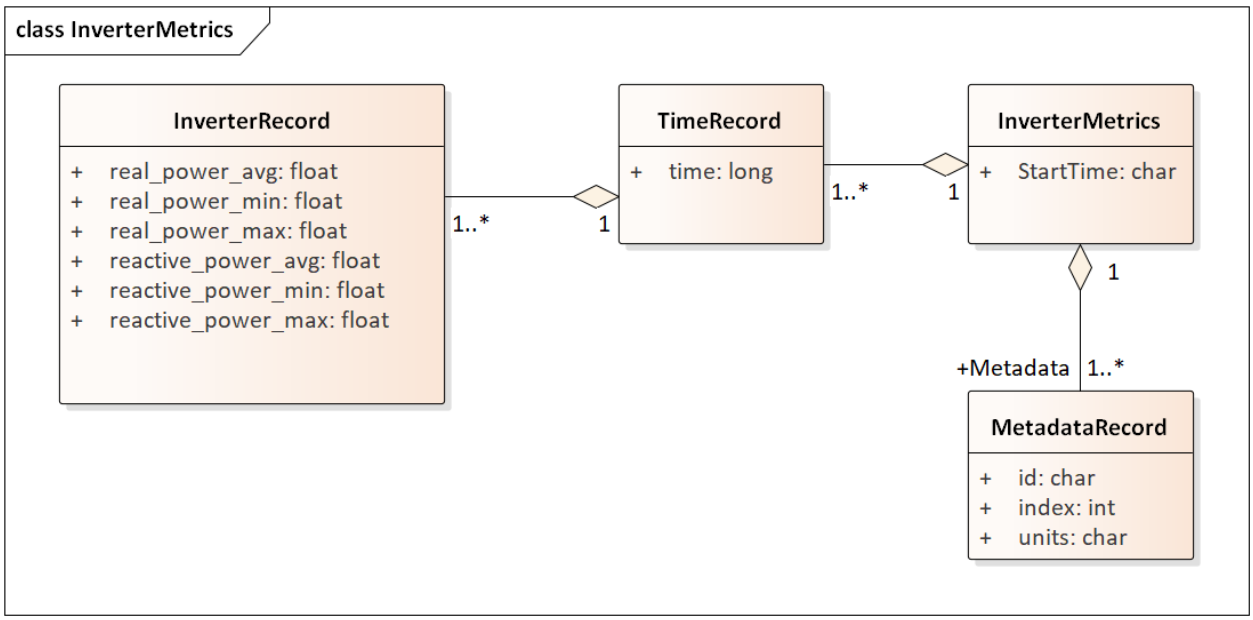


Fig. 5.16: GridLAB-D inverter metrics

(continued from previous page)

```
    },
    "real_power_avg": {
        "index": 2,
        "units": "W"
    },
    "real_power_max": {
        "index": 1,
        "units": "W"
    },
    "real_power_min": {
        "index": 0,
        "units": "W"
    }
},
"StartTime": "2013-07-01 00:00:00 PDT",
"10200": {
    "battery_inverter_house10_R1_12_47_1_tm_157": [
        -0.0,
        -0.0,
        0.0,
        0.0,
        0.0,
        0.0
    ],
    "battery_inverter_house10_R1_12_47_1_tm_273": [
        -0.0,
        -0.0,
        0.0,
        0.0,
```

(continues on next page)

(continued from previous page)

```
        0.0,
        0.0
    ],
    ...
    }
"10500": {
    "battery_inverter_house10_R1_12_47_1_tm_157": [
        -0.0,
        -0.0,
        0.0,
        0.0,
        0.0,
        0.0
    ],
    ...
    }
}
```

Fig. 5.17 shows the transactive agent dictionary and metrics file structures. Currently, these include one double-auction market per substation and one double-ramp controller per HVAC. Each dictionary (map) is keyed to the controller or market id. The Controller dictionary (top left) has a *houseName* for linkage to a specific house within the GridLAB-D model. In Version 1, there can be only one Market instance per GridLAB-D model, but this will expand in future versions. See the GridLAB-D market module documentation for information about the other dictionary attributes.

There will be two JSON metrics output files for TEAgents during a simulation, one for markets and one for controllers, which are structured as shown at the bottom of Fig. 5.17. The use of *StartTime* and Metadata is the same as for PYPOWER and GridLAB-D metrics. For controllers, the bid price and quantity (kw, not kwh) is recorded for each market clearing interval's id. For auctions, the actual clearing price and type are recorded for each market clearing interval's id. That clearing price applies throughout the feeder, so it can be used for supplemental revenue calculations until more agents are developed.

The EnergyPlus dictionary and metrics structure in Fig. 5.18 follows the same pattern as PYPOWER, GridLAB-D and TEAgent metrics. There are 42 metric values in the array, most of them pertaining to heating and cooling system temperatures and states. Each EnergyPlus model is custom-built for a specific commercial building, with detailed models of the HVAC equipment and zones, along with a customized Energy Management System (EMS) program to manage the HVAC. Many of the metrics are specified to track the EMS program performance during simulation. In addition, the occupants metric can be used for weighting the comfort measures; EnergyPlus estimates the number of occupants per zone based on hour of day and type of day, then TESP aggregates for the whole building. The *electric_demand_power* metric is the total three-phase power published to GridLAB-D, including HVAC and variable loads from lights, refrigeration, office equipment, etc. The *kwhr_price* will correspond to the market clearing price from Fig. 5.17. Finally, the *ashrae_uncomfortable_hours* is based on a simple standardized model, aggregated for all zones [4].

5.1.4 GridLAB-D Enhancements

The TSP simulation task includes maintenance and updates to GridLAB-D in support of TESP. This past year, the GridLAB-D enhancements done for TESP have included:

1. Extraction of the double-auction market and double-ramp controller into separate modules, with communication links to the internal GridLAB-D houses. This pattern can be reused to open up other GridLAB-D controller designs to a broader community of developers.

2. Porting the FNCS-enabled version of GridLAB-D to Microsoft Windows. This had not been working with the MinGW compiler that was recently adopted for GridLAB-D on Windows, and it will be important for other

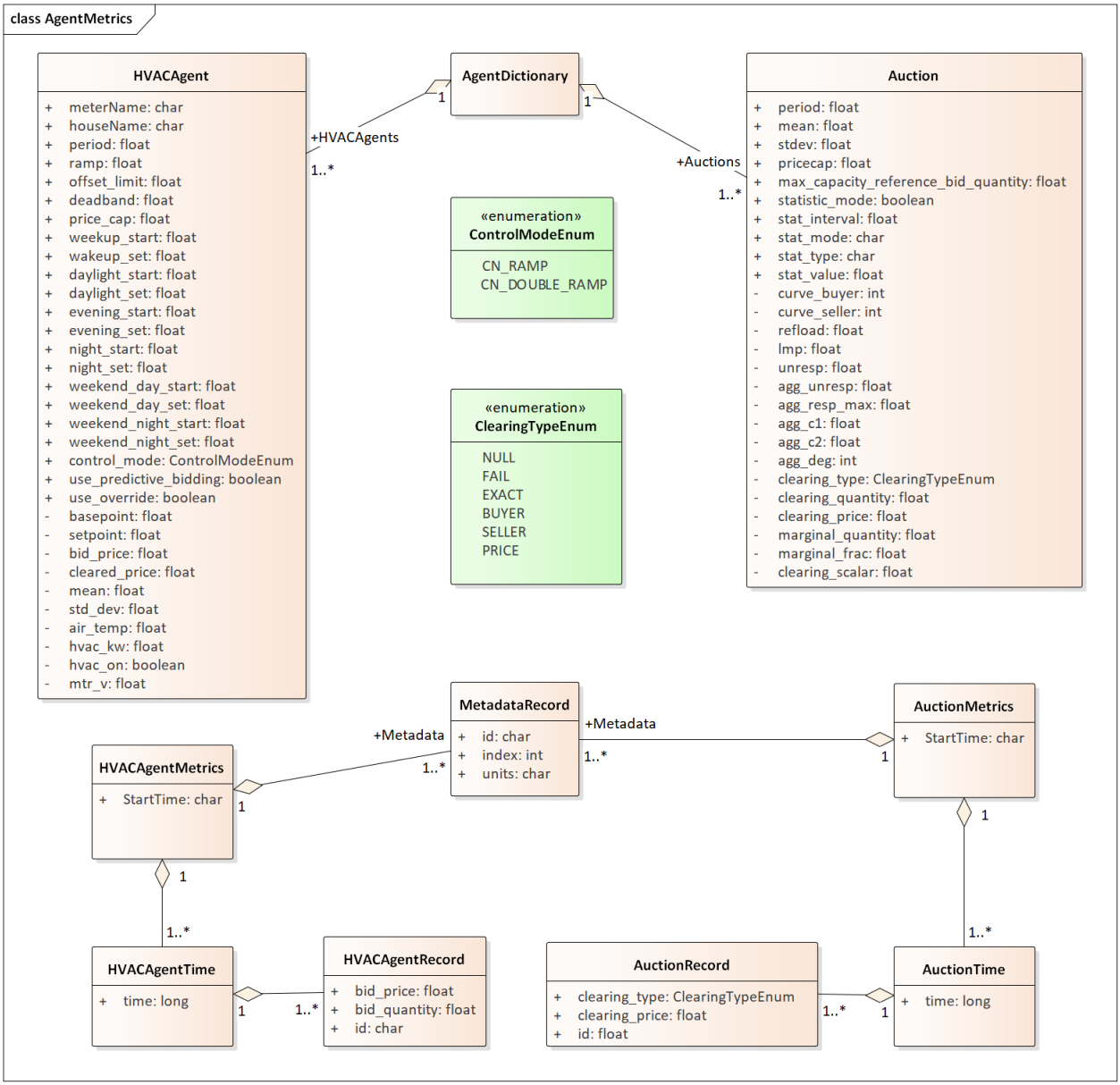


Fig. 5.17: TEAgent dictionary and metrics

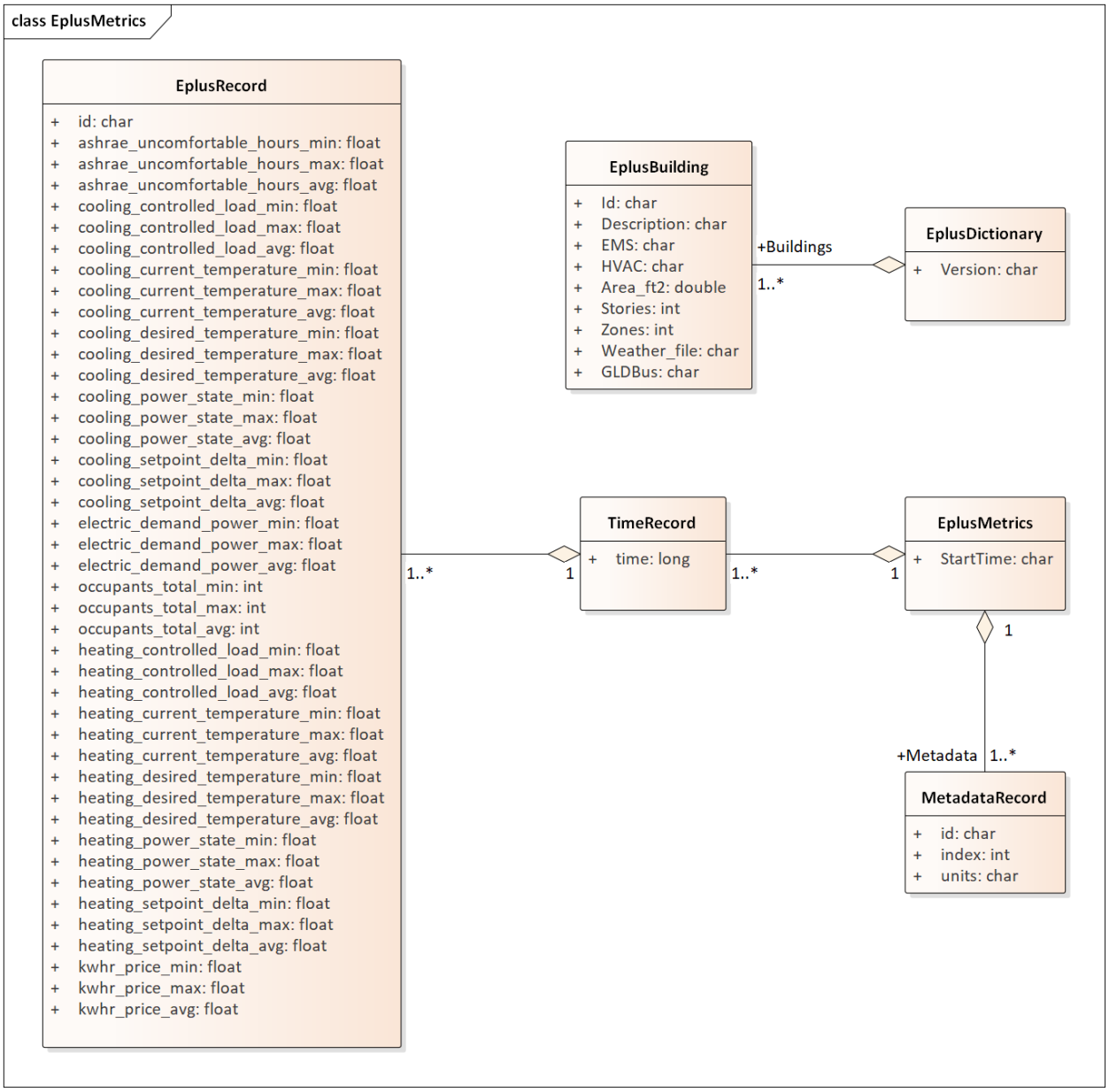


Fig. 5.18: EnergyPlus dictionary and metrics

projects.

3. Implementing the JSON metrics collector and writer classes in the tape module. This should provide efficiency and space benefits to other users who need to post-process GridLAB-D outputs.
4. Implementing a JSON-based message format for agents running under FNCS. Again, this should provide efficiency benefits for other projects that need more complicated FNCS message structures.

5.1.5 Developing Valuation Scripts

In order to provide new or customized valuation scripts in Python, the user should first study the provided examples. These illustrate how to load the JSON dictionaries and metrics described in Section 1.5, aggregate and post-process the values, make plots, etc. Coupled with some experience or learning in Python, this constitutes the easiest route to customizing TESP.

5.1.6 Developing Agents

The existing auction and controller agents provide examples on how to configure the message subscriptions, publish values, and link with HELICS at runtime. Section 1.4 describes the existing messages, but these constitute a minimal set for Version 1. It's possible to define your own messages between your own TEAgents, with significant freedom. It's also possible to publish and subscribe, or "peek and poke", any named object / attribute in the GridLAB-D model, even those not called out in Section 1.4. For example, if writing a waterheater controller, you should be able to read its outlet temperature and write its tank setpoint via HELICS messages, without modifying GridLAB-D code. You probably also want to define metrics for your TEAgent, as in Section 1.5. Your TEAgent will run under supervision of a HELICS broker program. This means you can request time steps, but not dictate them. The overall pattern of a HELICS-compliant program will be:

1. Initialize HELICS and subscribe to messages, i.e. notify the broker.
2. Determine the desired simulation *stop_time*, and any time step size (*delta_t*) preferences. For example, a transactive market mechanism on 5-minute clearing intervals would like *delta_t* of 300 seconds.
3. Set *time_granted* to zero; this will be under control of the HELICS broker.
4. Initialize *time_request*; this is usually *0 + delta_t*, but it could be *stop_time* if you just wish to collect messages as they come in.
5. While *time_granted* < *stop_time*:
   a. Request the next *time_request* from HELICS; your program then blocks.
   b. HELICS returns *time_granted*, which may be less than your *time_request*. For example, controllers might submit bids up to a second before the market interval closes, and you should keep track of these.
   c. Collect and process the messages you subscribed to. There may not be any if your time request has simply come up. On the other hand, you might receive bids or other information to store before taking action on them.
   d. Perform any supplemental processing, including publication of values through HELICS. For example, suppose 300 seconds have elapsed since the last market clearing. Your agent should settle all the bids, publish the clearing price (and other values), and set up for the next market interval.
   e. Determine the next *time_request*, usually by adding *delta_t* to the last one. However, if *time_granted* has been coming irregularly in 5b, you might need to adjust *delta_t* so that you do land on the next market clearing interval. If your agent is modeling some type of dynamic process, you may also adapt *delta_t* to the observed rates of change.
   f. Loop back to 5a, unless *time_granted* >= *stop_time*.

6. Write your JSON metrics file; Python has built-in support for this.
7. Finalize HELICS for an orderly shutdown, i.e. notify the broker that you're done.

The main points are to realize that an overall "while loop" must be used instead of a "for loop", and that the *time_granted* values don't necessarily match the *time_requested* values.

Developers working with C/C++ will need much more familiarity with compiling and linking to other libraries and applications, and much more knowledge of any co-simulators they wish to replace. This development process generally takes longer, which represents added cost. The benefits could be faster execution times, more flexibility in customization, code re-use, etc.

5.1.7 tesp_support Package Design

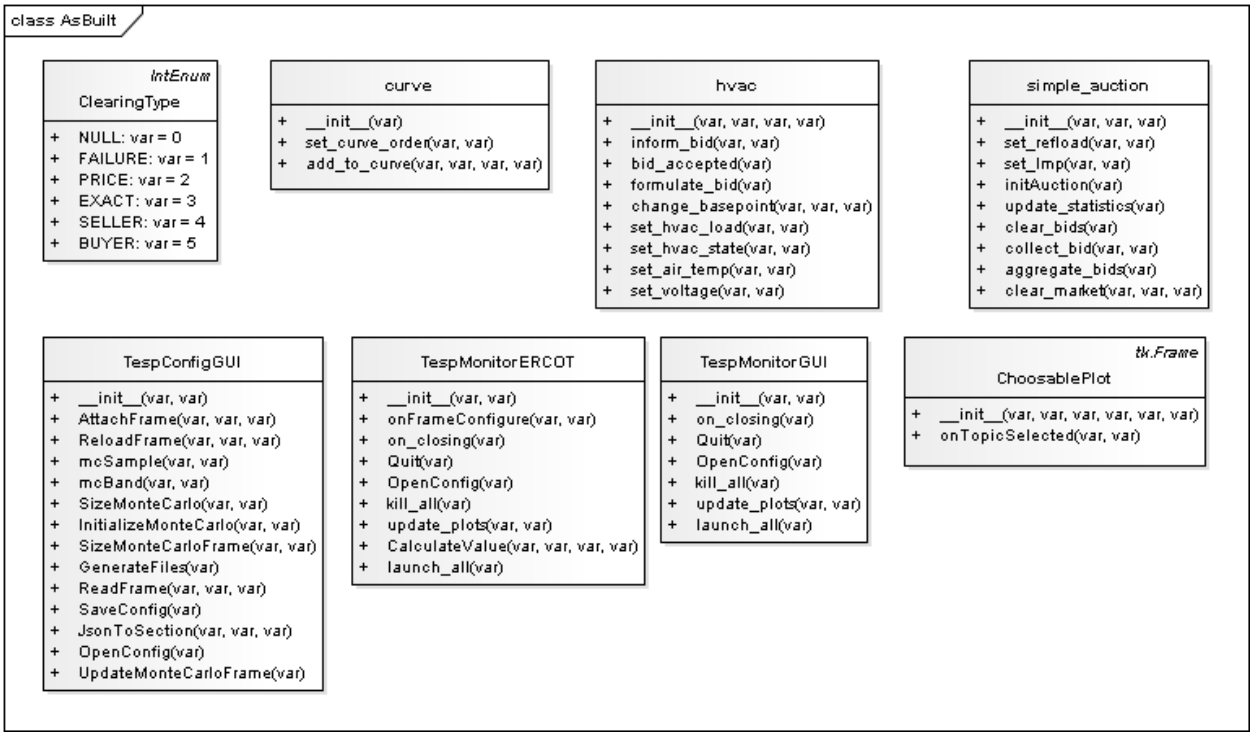


Fig. 5.19: Classes in the tesp_support package.

5.1.8 Development Work Flow for tesp_support

This is the main code repository for Python-based components of TESP, including the transactive agents, case configuration and post-processing. Currently, there are three kinds of transactive agent implemented here:

1. double-auction spot market, typically runs every 5 to 15 minutes
2. an electric cooling controller based on the Olympic Peninsula double-ramp method
3. an electric pre-cooling controller used to mitigate overvoltages in the NIST TE Challenge Phase 2

To develop a new agent, you may choose to copy an example Python file from this directory into your own test directory, to serve as a starting point. When finished, you should integrate the agent into this tesp_support package, so it will be available to other TESP developers and users. In this re-integration process, you also need to modify api.py so that

other Python code can call your new agent, and test it that way before re-deploying tesp_support to PyPi. Also review setup.py in the parent directory to make sure you've included any new dependencies, including version updates.

A second method is to create your new file(s) in this directory, which integrates your new agent from the start. There will be some startup effort in modifying api.py and writing the script/batch files to call your agent from within your working test directory. It may pay off in the end, by reducing the effort and uncertainty of code integration at the end.

Suggested sequence of test cases for development:

1. 30-house example at https://github.com/pnnl/tesp/tree/master/examples/te30. This includes one large building, one connection to a 9-bus/4-generator bulk system, and a stiff feeder source. The model size is suited to manual adjustments, and testing the interactions of agents at the level of a feeder or lateral. There are effectively no voltage dependencies or overloads, except possibly in the substation transformer. This case runs on a personal computer in a matter of minutes.
2. 8-bus ERCOT example at https://github.com/pnnl/tesp/tree/master/ercot/case8. This includes 8 EHV buses and 8 distribution feeders, approximately 14 bulk system units, and several thousand houses. Use this for testing your agent configuration from the GridLAB-D metadata, for large-scale interactions and stability, and for interactions with other types of agent in a less controllable environment. This case runs on a personal computer in a matter of hours.
3. 200-bus ERCOT example, when available. This will have about 600 feeders with several hundred thousand houses, and it will probably have to run on a HPC. Make sure the code works on the 30-house and 8-bus examples first.

- From this directory, 'pip install -e .' points Python to this cloned repository for any calls to tesp_support functions
- See the https://github.com/pnnl/tesp/tree/master/src/tesp_support/tesp_support for a roadmap of existing Python source files, and some documentation. Any changes or additions to the code need to be made in this directory.
- Run tests from any other directory on this computer
- When ready, edit the tesp_support version number and dependencies in setup.py
- To deploy, 'python setup.py sdist upload'
- Any user gets the changes with 'pip install tesp_support –upgrade'
- Use 'pip show tesp_support' to verify the version and location on your computer

5.2 Code Reference

5.2.1 TSO Case Data

The TSO schema was based on the MATPOWER formats for the network and generator cost data, supplemented with TESP data. Code in *tso_PYPOWER.py and tso_psst.py* reads this data from a JSON file.

Transmission System Operator (TSO)

http://example.com/root.json		
type	*object*	
properties		
• **version**	*The Version Schema*	
	not used	
	type	*integer*

continues on next page

Table 5.1 – continued from previous page

	examples	2		
	default	0		
• **baseMVA**	*The Basemva Schema*			
	MVA base for impedances			
	type	*integer*		
	examples	100		
	default	0		
• **StartTime**	*The Starttime Schema*			
	Date and time corresponding to 0 seconds in simulation			
	type	*string*		
	examples	2013-07-01 00:00:00		
	pattern	^(.*)$		
	default			
• **Tmax**	*The Tmax Schema*			
	Number of seconds to simulate			
	type	*integer*		
	examples	86400		
	default	0		
• **Period**	*The Period Schema*			
	Optimal power flow (OPF) interval in seconds			
	type	*integer*		
	examples	300		
	default	0		
• **dt**	*The Dt Schema*			
	Regular power flow (PF) interval in seconds			
	type	*integer*		
	examples	60		
	default	0		
• **pf_dc**	*The Pf_dc Schema*			
	1 for DC PF, 0 for AC PF			
	type	*integer*		
	examples	1		
	default	0		
• **opf_dc**	*The Opf_dc Schema*			
	1 for DC OPF, 0 for AC OPF			
	type	*integer*		
	examples	1		
	default	0		
• **bus**	*The Bus Schema*			
	Bus data, including loads and voltage base			
	type	*array*		
	items	*Bus Array*		
		type	*array*	
		items		
		•	*Bus Number*	
			type	*number*
			examples	1
		•	*Type (1=load,2=gen,3=swing)*	
			type	*number*
			enum	1, 2, 3
			examples	3
		•	*Pd (load)*	

continues on next page

Table 5.1 – continued from previous page

				type	*number*
				examples	15167.5
			•	*Qd (load)*	
				type	*number*
				examples	3079.89
			•	*Gs (shunt MW)*	
				type	*number*
				examples	0
			•	*Bs (shunt MVA)*	
				type	*number*
				examples	5000
			•	*Area*	
				type	*number*
				examples	1
			•	*V magnitude (pu)*	
				type	*number*
				examples	1
			•	*V angle (deg)*	
				type	*number*
				examples	0
			•	*kV base*	
				type	*number*
				examples	345
			•	*Zone*	
				type	*number*
				examples	1
			•	*Vmax pu*	
				type	*number*
				examples	1.1
			•	*Vmin pu*	
				type	*number*
				examples	0.9
• **gen**	*The Gen Schema*				
	Generator ratings				
	type	*array*			
	items	*Generator Array*			
		type	*array*		
		items			
			•	*Bus*	
				type	*number*
				examples	1
			•	*Pg (MW)*	
				type	*number*
				examples	0
			•	*Qg (MVAR)*	
				type	*number*
				examples	0
			•	*Qmax (MVAR)*	
				type	*number*
				examples	6567
			•	*Qmin (MVAR)*	
				type	*number*

continues on next page

Table 5.1 – continued from previous page

		examples	-6567
	•	*Vg (pu)*	
		type	*number*
		examples	1
	•	*MVA base*	
		type	*number*
		examples	19978.8
	•	*Status (1 in service)*	
		type	*number*
		enum	0, 1
		examples	1
	•	*Pmax (MW)*	
		type	*number*
		examples	19978.8
	•	*Pmin (MW)*	
		type	*number*
		examples	1998
	•	*Pc1*	
		type	*number*
		examples	0
	•	*Pc2*	
		type	*number*
		examples	0
	•	*Qc1min*	
		type	*number*
		examples	0
	•	*Qc1max*	
		type	*number*
		examples	0
	•	*Qc2min*	
		type	*number*
		examples	0
	•	*Qc2max*	
		type	*number*
		examples	0
	•	*AGC ramp rate*	
		type	*number*
		examples	0
	•	*10-min ramp rate*	
		type	*number*
		examples	0
	•	*30-min ramp rate*	
		type	*number*
		examples	0
	•	*Reactive ramp rate*	
		type	*number*
		examples	0
	•	*Area participation factor*	
		type	*number*
		examples	0
• **branch**	*The Branch Schema*		
	Lines and transformers; pu impedance and ratings		

continues on next page

Table 5.1 – continued from previous page

	type	*array*		
	items	*Branch Array*		
		type	*array*	
		items		
		•	*From Bus*	
			type	*number*
			examples	5
		•	*To Bus*	
			type	*number*
			examples	6
		•	*R (pu)*	
			type	*number*
			examples	0.004237
		•	*X (pu)*	
			type	*number*
			examples	0.035898
		•	*B (pu)*	
			type	*number*
			examples	2.48325
		•	*Rating A, short term (MVA)*	
			type	*number*
			examples	2168
		•	*Rating B, long term (MVA)*	
			type	*number*
			examples	2168
		•	*Rating C, emergency (MVA)*	
			type	*number*
			examples	2168
		•	*Tap Ratio for Xfmrs (From/To)*	
			type	*number*
			examples	0
		•	*Shift Angle for Xfmrs (deg)*	
			type	*number*
			examples	0
		•	*Status (1 in service)*	
			type	*number*
			enum	0, 1
			examples	1
		•	*Min Angle Difference From-To (deg)*	
			type	*number*
			examples	-360
		•	*Max Angle Difference From-To (deg)*	
			type	*number*
			examples	360
• **areas**	*The Areas Schema*			
	PF areas are not currently used in TESP			
	type	*array*		
• **gencost**	*The Gencost Schema*			
	Cost functions for generators and dispatchable loads			
	type	*array*		
	items	*Generator Cost Array*		
		Indexing must match the Generators		

continues on next page

Table 5.1 – continued from previous page

		type	*array*	
		items		
		•	*Flag - 2 for polynomial, 1 for piecewise linear*	
			type	*number*
			examples	2
		•	*Startup cost*	
			type	*number*
			examples	0
		•	*Shutdown cost*	
			type	*number*
			examples	0
		•	*Number of coefficients*	
			type	*number*
			examples	3
		•	*C2 coefficient*	
			type	*number*
			examples	0.005
		•	*C1 coefficient*	
			type	*number*
			examples	40
		•	*C0 coefficient*	
			type	*number*
			examples	0
• **DSO**	*The DSO Schema*			
	DSO topics, scaling factors and initial conditions at selected buses			
	type	*array*		
	items	*DSO Bus Array*		
		type	*array*	
		items		
		•	*bus ID*	
			type	*integer*
			examples	1
		•	*Name for passing messages*	
			type	*string*
			examples	SUBSTATION1
		•	*GridLAB-D Scale Factor*	
			type	*number*
			examples	792
		•	*Nominal P in MW*	
			type	*number*
			examples	15167.5
		•	*Nominal Q in MVAR*	
			type	*number*
			examples	3079.89
		•	*Scale factor for curve load*	
			type	*number*
			examples	0.5
		•	*Skew for curve load in seconds*	
			type	*number*
			examples	1711
		•	*Estimated P at time 0*	

continues on next page

Table 5.1 – continued from previous page

			type	*number*
			examples	4788.99
		•	*Estimated Q at time 0*	
			type	*number*
			examples	972.66
• **UnitsOut**	*The Units Out Schema*			
	Schedule of generators out of service			
	type	*array*		
	items	*Generator Outage Array*		
		type	*array*	
		items		
		•	*Index into Generators*	
			type	*number*
			examples	1
		•	*Time outage starts in seconds*	
			type	*number*
			examples	108000
		•	*Time outage ends in seconds*	
			type	*number*
			examples	154000
• **BranchesOut**	*The Branches Out Schema*			
	Schedules of branches out of service			
	type	*array*		
	items	*Branch Outage Array*		
		type	*array*	
		items		
		•	*Index into Branches*	
			type	*number*
			examples	2
		•	*Time outage starts in seconds*	
			type	*number*
			examples	108000
		•	*Time outage ends in seconds*	
			type	*number*
			examples	154000
• **swing_bus**	*The Swing_bus Schema*			
	Swing bus designation, depends on unit commitment			
	type	*integer*		
	examples	1		
	default	0		
definitions				

5.2.2 src Directory Structure

This list shows **directories** and *Python files* under the **tesp/src** repository. On GitHub, each README contains a list of other files.

- **archive**
  - **pypower**; legacy files to patch PYPOWER; we have been able to incorporate these patches into the main PYPOWER distribution.
- **comms_metrics**; a Power Distribution System Model Analysis tool not yet public though pypi. Not only can it perform metric calculations, it also has the ability to plot the models as a network and parse different file formats as pre-processing for the data analysis.
- **energyplus**; C++ code to build a simple interface agent for EnergyPlus; this is part of the TESP distribution and used in the te30, sgip1 and energyplus examples.
- **gridlabd**; legacy files for the house populations and feeder growth model; these features are mostly subsumed into tesp_support
- **jupyter**; a prototype Jupyter notebook used for post-processing demonstrations and training
- **matpower**
  - **ubuntu**; legacy code that wraps MATPOWER for TESP, but only on Ubuntu. We now use PYPOWER. In 2017, the wrapping process was very difficult on Mac OS X, and unsuccessful on Windows using free compilers.
- **synComGraph**; graph algorithms to generate a synthetic communication network graph topology corresponding to a given feeder topology
- **tesp_support**; utilities for building and running using PYPOWER with or without FNCS/HELICS co-simulations
  - *setup.py*; contains the version number and dependencies for tesp_support package
  - **tesp_support**; Python code for agents, configuration and post-processing
    - **api; code that configures new capabilities for TESP**
      - *data.py*; the paths to data libraries
      - *entity.py*; utilities for assign json file to attribute in python script
      - *fncs.py*; the Python interface to FNCS, which is a C/C++ shared object library, or dynamic link library (Windows)
      - *helpers.py*; utility functions for use within tesp_support
      - *make_ems.py*; creates and merges the EMS for an EnergyPlus building model
      - *model.py*; GridLAB-D model I/O for TESP api
      - *modifier.py*; modify GridLAB-D model I/O for TESP api
      - *metric_api.py*; utility metric api functions for use in post-processing
      - *metric_collector.py*; utility metric collector functions for use within simulation or post process
      - *parse_helpers.py*; parse text for different types of numbers
      - *player.py*; configure and plays a files for a simulation
      - *process_eplus.py*; makes tabular and plotted summaries of EnergyPlus results
      - *process_gld.py*; makes tabular and plotted summaries of GridLAB-D results (substation power/losses, average and sample house temperatures, meter voltage min/max)

- *process_houses.py*; plots the HVAC power and air temperature for all houses
- *process_inv.py*; makes tabular and plotted summaries of results for NIST TE Challenge 2, including inverters, capacitor switching and tap changes
- *process_pypower.py*; makes tabular and plotted summaries of PYPOWER results for the 9-bus model in te30 or sgip1
- *process_voltages.py*; plots the minimum and maximum voltage for all houses
- *test_runner.py*; auto test runner for TESP run* cases based on pre-existing shell script file.
- *time_helpers.py*; utility time functions for use within tesp_support, including new agents
- *tso_helpers.py*; helpers for PYPOWER, PSST, MOST solutions
- *tso_PSST.py*; manages PSST solutions for the DSOT example, based on a 8-bus or 200-bus model. Note that the ERCOT cases use custom local versions of this code instead.
- *tso_PYPOWER.py*; manages PYPOWER solutions for the te30 and sgip1 examples, based on a 9-bus textbook model. Note that the ERCOT cases use custom local versions of this code instead.

* **original; legacy code that configures most example/capabilities for TESP**
  - *commercial_feeder_glm.py*; from a PNNL taxonomy feeder as the backbone, populates it with commercial building, solar PV, batteries and smart inverters
  - *copperplate_feeder_glm.py*; from a PNNL taxonomy feeder as the backbone, populates it with sudo copperplate
  - *curve*; accumulates a set of price, quantity bids for later aggregation for a curve
  - *glm_dict.py*; parses the GridLAB-D input (GLM) file and produces metafile data in JSON format, describing the houses, meters, DER, capacitors and regulators
  - *precool.py*; manages a set of house thermostats for NIST TE Challenge 2. There is no communication with a market. If the house experiences an overvoltage, the thermostat is turned down and locked for 4 hours, unless the house temperature violates comfort limits.
  - *prep_precool.py*; configures the agent metadata (JSON) and GridLAB-D HELICS subscriptions/publications for NIST TE Challenge 2 precooling
  - *prep_substation.py*; configures the agent metadata (JSON) and GridLAB-D HELICS subscriptions/publications for the double-auction, double-ramp simulations
  - *process_agents.py*; makes tabular and plotted summaries of agent results
  - *residential_feeder_glm.py*; from a PNNL taxonomy feeder as the backbone, populates it with houses, solar PV, batteries and smart inverters
  - *simple_auction.py*; implements the double-auction agent and the Olympic Peninsula cooling agent, as separate Python classes, called by auction.py
  - *tesp_case.py*; supervises the assembly of a TESP case with one feeder, one EnergyPlus building and one PYPOWER model. Reads the JSON file from tesp_config.py
  - *tesp_config.py*; a GUI for creating the JSON file used to configure a TESP case
  - *tesp_monitor.py*; a GUI for launching a TESP simulation, monitoring its progress, and terminating it early if necessary

* **weather; code that configures weather capabilities for TESP**
  - *PSM_download.py*; simple script to download PSM weather files and convert them to DAT files

        - *PSMv3toDAT.py*; this code reads in PSM v3 csv files to converts weather DAT format for common use by agents
        - *README.md*; this file
        - *TMY3toCSV.py*; converts TMY3 weather data to CSV format for common use by agents
        - *TMYtoEPW.py*; command-line script that converts a TMY2 file to the EnergyPlus EPW format
        - *weather_Agent.py*; publishes weather and forecasts based on a CSV file
    - **consensus**; custom code that for running the consensus mechanism on microgrid n DSOT co simulation using TSO and DSO DER agents.
    - **dsot**; custom code that for running the DSOT co simulation using TSO and DSO DER agents. Used for a 2021 journal paper on TESP and the DSOT example.
    - **sgip1**; custom code that plotted curves from different cases on the same graph. Used for a 2018 journal paper on TESP and the SGIP1 example.
    - **matpower**; legacy code that configures and post-processes MATPOWER v5+ for TESP. We now use PYPOWER and PSST instead.
    - **valuation**; custom code that post-processed SGIP1 outputs for the 2018 journal paper. May serve as an example, or use Jupyter notebooks instead.
  - **test**; scripts that support testing the package; not automated

5.2.3 Links to Dependencies

- Docker
- EnergyPlus
- GridLAB-D
- Matplotlib
- MATPOWER
- NetworkX
- NumPy
- Pandas
- pip
- PYPOWER
- Python
- SciPy
- TESP

5.3 tesp_support package

Transactive Energy Simulation Platform (TESP) Contains the python packages for the tesp_support

Example

To start PYPOWER for connection to FNCS:

```
import tesp_support.original.tso_PYPOWER_f as tesp
tesp.tso_pypower_loop_f('te30_pp.json','TE_Challenge')
```

To start PYPOWER for connection to HELICS:

```
import tesp_support.api.tso_PYPOWER as tesp
tesp.tso_pypower_loop('te30_pp.json','TE_Challenge', helicsConfig='tso.json')
```

5.3.1 Subpackages

tesp_support.api package

Transactive Energy Simulation Platform (TESP) Contains the python files for any analysis

Submodules

tesp_support.api.bench_profile module

`tesp_support.api.bench_profile.`**`bench_profile`**(*func*)

tesp_support.api.data module

tesp_support.api.entity module

class `tesp_support.api.entity.`**`Entity`**(*entity*, *config*)

Bases: `object`

`add_attr`(*datatype*, *label*, *unit*, *item*, *value=None*)

Add the Item attribute to the Entity

Parameters

- **`datatype`** (*str*) – Describes the datatype of the attribute
- **`label`** (*str*) – Describes the attribute
- **`unit`** (*str*) – The unit name of the attribute
- **`item`** (*str*) – The name of the attribute
- **`value`** (*any*) – The value of the item

Return type

Item

`count()`

Returns
The number of defined Items in the Entity

Return type
int

`del_attr(item)`

Delete the Item from the Entity

Parameters
item (*str*) – name of the attribute in the Entity

`del_instance(object_name)`

Delete the Entity instance

Parameters
object_name (*str*) – the name of the instance

`del_item(object_name, item)`

Delete the value of the Entity instance from the Item

Parameters

- **object_name** (*str*) – the name of the instance
- **item** (*str*) – name of the Item

`find_item(item)`

Find the Item from the Entity

Parameters
item (*str*) – name of the attribute in the entity

Return type
Item

`get_instance(object_name)`

Get the Entity instance

Parameters
object_name (*str*) – the name of the instance

Returns
an object with name and values or None when object_name is invalid

Return type
Entity instance

`instanceToJson()`

Stringify the instance(s) in the Entity to JSON

Returns
JSON string of the instance(s) in the Entity

Return type
str

`instanceToSQLite(connection)`

Commit the instance(s) in the Entity to SQLite

Parameters
connection – A valid sqlite connection object

set_instance(*object_name*, *params*)

Set the Entity instance the given set of parameters

Parameters

- **object_name** (*str*) – the name of the instance
- **params** (*list<list>*) – list of the attribute parameters

Returns
an object with name and values

Return type
Entity instance

set_item(*object_name*, *item*, *val*)

Set the value of the Entity instance for the Item

Parameters

- **object_name** (*str*) – the name of the instance
- **item** (*str*) – name of the Item
- **val** (*any*) – value of the item

Returns
the value or None when the value has not been set

Return type
any

toHelp()

List the Item(s) in the Entity in help format with datatype, label, name, default value

Returns
format list of the Items in the Entity

Return type
str

toJson()

List the Item(s) in the Entity in json string format

Returns
JSON string of the Items in the Entity

Return type
str

toList()

List the Item(s) in the Entity

Returns
list of Items in the Entity

Return type
dict

toSQLite(*connection*)

Create a sqlite table to store the Item(s) in the Entity
with datatype, label, name, unit, default value

Parameters
connection (*Connection*) – A valid sqlite connection object

class tesp_support.api.entity.**Item**(*datatype*, *label*, *unit*, *item*, *value=None*, *range_check=None*)

Bases: object

toFrame()

List the attribute in the Items

Returns
with label, value, unit, datatype, name, range_check

Return type
dist

toJSON()

Stringify the attribute in the Items to JSON

Returns
JSON string with label, unit, datatype, value

Return type
str

toList()

List the attribute in the Items

Returns
with label, value, unit, datatype, name

Return type
dist

tesp_support.api.entity.**assign_defaults**(*obj*, *file_name*)

Utilities that opens a JSON file and assigns the attributes to the specified object

Parameters

- **obj** (*object*) – any object like module or class
- **file_name** (*str*) – a JSON file fully qualified path and name

Returns
a dictionary of the JSON that has been loaded

Return type
dict

tesp_support.api.entity.**assign_item_defaults**(*obj*, *file_name*)

Utilities that opens a JSON file and assigns the attributes Item to the specified object

Parameters

- **obj** (*object*) – any object like module or class
- **file_name** (*str*) – a JSON file fully qualified path and name

Returns
a dictionary of the JSON that has been loaded

Return type
dict

tesp_support.api.gridpiq module

tesp_support.api.helpers module

Utility functions for use within tesp_support, including new agents.

class tesp_support.api.helpers.**HelicsMsg**(*name*, *period*)

Bases: object

config(*_n*, *_v*)

pubs(*_g*, *_k*, *_t*, *_o*, *_p*)

pubs_e(*_g*, *_k*, *_t*, *_u*)

pubs_n(*_g*, *_k*, *_t*)

subs(*_k*, *_t*, *_o*, *_p*)

subs_e(*_r*, *_k*, *_t*, *_i*)

subs_n(*_k*, *_t*)

write_file(*_fn*)

class tesp_support.api.helpers.**all_but_one_level**(*level*)

Bases: object

static filter(*logRecord*)

class tesp_support.api.helpers.**all_from_one_level_down**(*level*)

Bases: object

filter(*logRecord*)

tesp_support.api.helpers.**enable_logging**(*level*, *model_diag_level*, *name_prefix*)

Enable logging for process

Parameters

- **level** (*str*) – the logging level you want set for the process
- **model_diag_level** (*int*) – initial value used to filter logging files
- **name_prefix** (*str*) – description prefix for the log file name

tesp_support.api.helpers.**get_run_solver**(*name*, *pyo*, *model*, *solver*)

tesp_support.api.helpers.**gld_strict_name**(*val*)

Sanitizes a name for GridLAB-D publication to FNCS GridLAB-D name should not begin with a number, or contain '-' for FNCS

Parameters
val (*str*) – the input name

Returns
val with all '-' replaced by '_', and any leading digit replaced by 'gld_'

Return type
str

`tesp_support.api.helpers.random_norm_trunc(dist_array)`

`tesp_support.api.helpers.zoneMeterName(ldname)`

Enforces the meter naming convention for commercial zones The commercial zones must be children of load objects This routine replaces "_load_" with "_meter".

Parameters
ldname (*str*) – the GridLAB-D name of a load, ends with _load_##

Returns
The GridLAB-D name of upstream meter

Return type
str

tesp_support.api.make_ems module

Creates and merges the EMS for an EnergyPlus building model

Public Functions:

make_ems
Creates the energy management system (EMS) for FNCS/HELICS to interface with EnergyPlus

merge_idf
Assembles the base IDF, the EMS, start time and end time

`tesp_support.api.make_ems.cooling_coil_sensor(name, target, op)`

`tesp_support.api.make_ems.get_eplus_token(sval)`

`tesp_support.api.make_ems.global_variable(name, op)`

`tesp_support.api.make_ems.heating_coil_sensor(name, target, op)`

`tesp_support.api.make_ems.idf_int(val)`

Helper function to format integers for the EnergyPlus IDF input data file

Parameters
val (*int*) – the integer to format

Returns
the integer in string format, padded with a comma and zero or one blanks, in order to fill three spaces

Return type
str

`tesp_support.api.make_ems.make_ems(sourcedir='./output', baseidf='SchoolBase.idf', target='ems.idf', write_summary=False, bHELICS=False)`

Creates the EMS for an EnergyPlus building model

Parameters

- **sourcedir** (*str*) – directory of the output from EnergyPlus baseline simulation, default ./output
- **baseidf** (*str*) – is the original EnergyPlus model file without the EMS
- **target** (*str*) – desired output file in PWD, default ems.idf
- **write_summary** –
- **bHELICS** –

`tesp_support.api.make_ems.`**`merge_idf`**(*base*, *ems*, *StartTime*, *EndTime*, *target*, *StepsPerHour*)

Assembles a base EnergyPlus building model with EMS and simulation period

Parameters

- **base** (*str*) – fully qualified base IDF model
- **ems** (*str*) – fully qualified EMS model file
- **StartTime** (*str*) – Date-Time to start simulation, Y-m-d H:M:S
- **EndTime** (*str*) – Date-Time to end simulation, Y-m-d H:M:S
- **target** (*str*) – fully qualified path for new model
- **StepsPerHour** –

`tesp_support.api.make_ems.`**`output_variable`**(*name*, *target*, *op*)

`tesp_support.api.make_ems.`**`print_idf_summary`**(*zones*, *zonecontrols*, *thermostats*, *schedules*, *hcoils*, *ccoils*, *hvacs*)

`tesp_support.api.make_ems.`**`schedule_actuator`**(*name*, *target*, *op*)

`tesp_support.api.make_ems.`**`schedule_sensor`**(*name*, *op*)

`tesp_support.api.make_ems.`**`summarize_idf`**(*fname*, *baseidf*)

`tesp_support.api.make_ems.`**`valid_var`**(*name*)

`tesp_support.api.make_ems.`**`write_new_ems`**(*target*, *zones*, *zonecontrols*, *thermostats*, *schedules*, *hcoils*, *ccoils*, *hvacs*, *bHELICS*)

`tesp_support.api.make_ems.`**`zone_heating_sensor`**(*name*, *op*)

`tesp_support.api.make_ems.`**`zone_occupant_sensor`**(*name*, *op*)

`tesp_support.api.make_ems.`**`zone_sensible_cooling_sensor`**(*name*, *op*)

`tesp_support.api.make_ems.`**`zone_sensible_heating_sensor`**(*name*, *op*)

`tesp_support.api.make_ems.`**`zone_temperature_sensor`**(*name*, *op*)

tesp_support.api.metrics_api module

`tesp_support.api.metrics_api.actual_der_vs_projected_ratio(actual_der, actual_col_name, projected_col_name, projected_der=None)`

This function calculates the accuracy of the predictive model, by comparing predicted results with actual results

Metric defined in the document VM_Actual Benefits_Predicted Benefits.docx

Parameters

- **actual_der** (*dataframe*) – time series dataframe that contains the total benefits from DER, as observed ex post.
- **actual_col_name** (*str*) – id of the dataframe column that contains the actual DER benefit values
- **projected_col_name** (*str*) – id of the dataframe column that contains the projected DER benefit values
- **projected_der** (*dataframe*) – time series dataframe that contains the projected benefits from DER, as observed ex post.

Returns
time series dataframe that contains the calculated ratios

Return type
dataframe

`tesp_support.api.metrics_api.get_average_air_temp_deviation(actual_df, actual_col_name, set_point_col_name, set_points_df, start_date_time)`

Function calculates per device average deviation from desired indoor temperature set point in a year for each DSO

Metric defined in document VM_Average Indoor Air Temp Deviation.docx

Parameters

- **actual_df** (*dataframe*) – per-device average deviation from desired air temperature set point
- **actual_col_name** (*str*) – dataframe column id for the location of actual temperatures
- **set_point_col_name** (*str*) – dataframe column id for the location of set point data
- **set_points_df** (*dataframe*) – time series data frame containing the set points data.
- **start_date_time** (*str*) – the starting date and time when the calculation should start

Returns

the average of the calculated differences between the average of actual
indoor temperature deviation from set point over one year.

Return type
float

`tesp_support.api.metrics_api.get_average_unit_price(time_series, column_id, start_date_time)`

Function calculates the market average unit price (of electricity) over the course of 8,760 hours, in a specific service territory managed by an independent system operator

Metric defined in document VM_Market Average Unit Price.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the hourly market electricity prices for a year
- **column_id** (*str*) – name of the dataframe column that the price data is located
- **start_date_time** (*str*) – the date and time that the calculations are to start at

Returns
the average unit price for the year

Return type
float

`tesp_support.api.metrics_api.get_avg_customer_demand(time_series, start_date, val_col_id)`

This function calculates the average of customer demand based on 8,760 hours of the year

Metric defined in VM_Average Customer Demand.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe representing a time series containing customer demand records
- **start_date** (*datetime*) – the start date and time that should be used in the calculation
- **val_col_id** (*str*) – id of the dataframe column which contains the customer demand data values

Returns
the calculated yearly average customer demand

Return type
float

`tesp_support.api.metrics_api.get_emergency_scarcity_sell(scarcity_power_df, scarcity_col_id, scarcity_price_df, price_col_id, generation_capacity_df, gen_col_id, available_power_df, available_col_id)`

Function calculates the annual value of firm energy for Scarcity Conditions

Metric is defined in the document VM_Emergency Scarcity Wholesales Sells.docx

Parameters

- **scarcity_power_df** (*dataframe*) – time series dataframe that contains the power data used in calculation
- **scarcity_col_id** (*str*) – name of the dataframe column where the power data is located
- **scarcity_price_df** (*dataframe*) – time series dataframe that contains the price data used in calculation
- **price_col_id** (*str*) – name of the dataframe column where the price data is located
- **generation_capacity_df** (*dataframe*) – time series dataframe that contains the generation data used in calculation
- **gen_col_id** (*str*) – name of the dataframe column where the generation data is located
- **available_power_df** (*dataframe*) – time series dataframe that contains the available power data used in calculation

- **available_col_id** (*str*) – name of the dataframe column where the available power data is located

Returns
time series dataframe containing the calculated scarcity values

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_feeder_energy_losses`**(*feeder_gen_df*, *gen_column_id*, *feeder_load_df*, *load_column_id*, *start_date_time*, *duration*)

Function calculates the impact of trans-active energy systems on feeder energy losses. Data records in the time series entered as input must be recorded at five minute intervals

Metric defined in document VM_Feeder Energy Losses.docx

Parameters

- **feeder_gen_df** (*dataframe*) – data frame containing the 5-min average total generation from bulk power system and DERs
- **gen_column_id** (*str*) – name of the column in the feeder generation dataframe where the generation data is located
- **feeder_load_df** (*dataframe*) – data frame containing the 5-min average total load
- **load_column_id** (*str*) – name of the column in the feeder load dataframe where the load data is located
- **start_date_time** (*str*) – calculation start date and time
- **duration** (*int*) – the duration of time in hours that the calculations are to be performed

Returns
a dataframe object containing the generation, load, and losses data

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_hot_water_deficit`**(*water_temperatures*, *water_column_id*, *desired_temperatures*, *desired_column_id*, *flow_rates*, *flow_column_id*, *delta_t*, *start_date_time*, *duration*)

Function calculates device energy deficit from desired hot water temperature set point in a year

Metric defined in document VM_Hot Water Supply Deficit.docx

Parameters

- **water_temperatures** (*dataframe*) – per device 5-min average hot water actual temperature
- **water_column_id** (*str*) – name of the dataframe column where the temperature data is located
- **desired_temperatures** (*dataframe*) – per device 5-min average hot water temperature set point
- **desired_column_id** (*str*) – name of the dataframe column where the set point temperature data is located
- **flow_rates** (*dataframe*) – per device 5-min average hot water flow rate

- **flow_column_id** (*str*) – name of the dataframe column where the flow rate data is located
- **delta_t** (*float*) – time difference
- **start_date_time** (*str*) – the date and time that the calculations are to start at
- **duration** (*int*) – the length of time which the calculations should be executed

Returns
time series dataframe containing the calculated deficit data

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_indoor_air_temp_deviation`**(*time_series*, *column_id*, *set_point*, *start_date_time*, *duration*)

Function calculates the maximum actual indoor temperature deviation from set point over one year.

Metric is defined in document VM_Max Indoor Air Temp Deviation.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains 5 minute max deviation data for a year
- **column_id** (*str*) – the name of the dataframe column where the deviation data is located
- **set_point** (*float*) – The set point value that is to be used in the calculation
- **start_date_time** (*str*) – the date and time that the calculations are to start
- **duration** (*int*) – the time duration in hours for which the calculations should be performed

Returns
time series dataframe containing the maximum deviations calculated hourly from the input data

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_max_comm_packet_size`**(*time_series*, *size_column_id*, *start_date_time*, *duration*)

Function calculates the maximum size of a message sent in the communication channels

Metric is defined in document VM_Maximum Communication Packet Size.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the communication network packet size Mbs
- **size_column_id** (*str*) – name of the dataframe column that contains the packet size data
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time which the calculations should be executed

Returns
the maximum communication packet size for the time period entered

Return type
float

`tesp_support.api.metrics_api.`**`get_max_duration_over_voltage`**(*time_series*, *column_id*, *limit_val*, *start_date_time*, *duration*)

Function calculates the maximum duration of an over-voltage event reported at each feeder

Metric defined in document VM_Max Duration of Over-Voltage Violations.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the 5-minute three-phase voltage data
- **column_id** (*str*) – name of the dataframe column the voltage data is located
- **limit_val** (*float*) – threshold value used to compare voltage values against
- **start_date_time** (*str*) – calculation start date and time
- **duration** (*int*) – the duration of time in hours that the calculations are to be performed

Returns
hourly time series dataframe containing the calculated maximum duration of voltage violating under-voltage limit

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_max_duration_under_voltage`**(*time_series*, *column_id*, *limit_val*, *start_date_time*, *duration*)

Function calculates the maximum duration of an under-voltage event reported at each feeder

Metric defined in document VM_Max Duration of Under-Voltage Violations.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the 5-minute three-phase voltage data
- **column_id** (*str*) – name of the dataframe column the voltage data is located
- **limit_val** (*float*) – threshold value used to compare voltage values against
- **start_date_time** (*str*) – calculation start date and time
- **duration** (*int*) – the duration of time in hours that the calculations are to be performed

Returns
hourly time series dataframe containing the calculated maximum duration of voltage violating under-voltage limit

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_max_market_price`**(*time_series*, *column_id*, *start_date_time*)

Function calculates the highest market price (of electricity) over the course of 8,760 hours, in a specific service territory managed by an independent system operator

Metric defined in document VM_Highest Market Price.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the hourly market price for electricity within a territory served by an ISO or balancing authority, for each of the 8,760 hours per year.
- **column_id** (*str*) – name of the dataframe column where the market price data is located
- **start_date_time** (*str*) – the date and time that the calculations are to start at

Returns
the maximum market price value found in the market price dataset

Return type
float

`tesp_support.api.metrics_api.get_max_over_voltage(time_series, column_id, threshold_val, start_date_time, duration)`

Function calculates the maximum over-voltage deviation reported each hour in the feeder voltage data

Metric is defined in document VM_Max Over-Voltage Violations.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing feeder voltage data in 5 minute intervals
- **column_id** (*str*) – the name of the dataframe column where the voltage data is located
- **threshold_val** (*float*) – the maximum threshold data that is used to compare against the voltage data
- **start_date_time** (*str*) – calculation start date and time
- **duration** (*int*) – the duration of time in hours that the calculations are to be performed

Returns
time series dataframe containing the calculated hourly over voltage maximum values

Return type
dataframe

`tesp_support.api.metrics_api.get_max_under_voltage(time_series, column_id, threshold_val, start_date_time, duration)`

Function calculates the maximum over-voltage deviation reported each hour in the feeder voltage data

Metric is defined in document VM_Max Under-Voltage Violations.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing feeder voltage data in 5 minute intervals
- **column_id** (*str*) – the name of the dataframe column where the voltage data is located
- **threshold_val** (*float*) – the maximum threshold data that is used to compare against the voltage data
- **start_date_time** (*str*) – calculation start date and time
- **duration** (*int*) – the duration of time in hours that the calculations are to be performed

Returns
time series dataframe containing the calculated hourly over voltage maximum values

Return type
dataframe

`tesp_support.api.metrics_api.get_mean_absolute_percentage(actual_load_df, actual_col_id, forecasted_load_df, forecasted_col_id, start_date_time, duration)`

Function calculates the prediction accuracy of load forecasting methods.

Metric is defined in document VM_Mean Absolute Percentage (Load) Error.docx

Parameters

- **actual_load_df** (*dataframe*) – time series dataframe containing the actual load observed over a period of time
- **actual_col_id** (*str*) – name of the column where actual load data is located
- **forecasted_load_df** (*dataframe*) – time series dataframe containing the forecasted load observed over a period of time
- **forecasted_col_id** (*str*) – name of the column where forecasted load data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated ratios, the calculated average value

Return type
dataframe, float

`tesp_support.api.metrics_api.get_minimum_market_price(time_series, price_col_id, start_date_time)`

Function calculates the minimum market price (of electricity) over the course of 8,760 hours

Metric defined in document VM_Minimum Market Price.docx

Parameters

- **time_series** (*dataframe*) – Hourly market prices for electricity
- **price_col_id** (*str*) – name of the dataframe column where the price data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start

Returns
the minimum market price found in the data over the course of a year

Return type
float

`tesp_support.api.metrics_api.get_peak_demand(time_series, column_id, start_date_time)`

This function calculates the highest hourly electricity demand (MW) in the year of data contained in the dataframe

This metric is defined in document VM_PeakDemand or PeakSupply.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the demand values over the course of a year
- **column_id** (*str*) – name of the dataframe column where the demand data is located
- **start_date_time** (*str*) – calculation start date and time

Returns
maximum value identified in the dataframe column identified by column_id

Return type
float

`tesp_support.api.metrics_api.get_peak_supply(time_series, column_id, start_date_time)`

This function calculates the highest hourly electricity supply (MW) in the year of data contained in the dataframe

This metric is defined in document VM_PeakDemand or PeakSupply.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the supply values over the course of a year
- **column_id** (*str*) – name of the dataframe column where the supply data is located
- **start_date_time** (*str*) – calculation start date and time

Returns
maximum value identified in the dataframe column identified by column_id

Return type
float

`tesp_support.api.metrics_api.`**`get_pv_aep_valuation`**(*solar_irradiation*, *pv_system_area*, *pv_system_efficiency*)

Function calculates and estimate of the total annual power output from a PV system in the units of kWh

Metric defined in the document VM_PV Annual Energy Production.docx

Parameters

- **solar_irradiation** (*float*) – total solar irradiation incident on PV surface in the units of kWh/sq.m.
- **pv_system_area** (*float*) – PV System Area
- **pv_system_efficiency** (*float*) – PV System Efficiency

Returns
the product of the three input values

Return type
float

`tesp_support.api.metrics_api.`**`get_reactive_power_demand`**(*time_series*, *max_col_id*, *avg_col_id*, *start_date_time*, *duration*)

Function calculates the maximum and average substation reactive power flow reported each hour

Metric is defined in document VM_Substation Reactive Power Demand.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the 5-minute substation reactive power flow data
- **max_col_id** (*str*) – name of the dataframe column containing the 5-minute maximum data
- **avg_col_id** (*str*) – name of the dataframe column containing the 5-minute average data
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the hourly maximum and average values calculated by the function

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_substation_peak_power`**(*time_series*, *power_col_id*, *start_date_time*, *duration*)

Function calculates a substation's maximum real power flow

Metric defined in document VM_Substation Peak Real Power Demand.docx

Parameters

- **time_series** (*dataframe*) – time series containing substation real power flow (Mvar) at 5 minute intervals
- **power_col_id** (*str*) – name of the dataframe column containing the power flow data
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated hourly peak power flow

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_synch_date_range`**(*time_series*)

Function returns the latest starting date/time and the earliest ending date/time of the time series data frames in the time series list

Parameters
time_series (*list<dataframe>*) – List containing a set of pandas dataframes each representing a time series

Returns
the latest start and the earliest end times found in the list of data frames

Return type
datetime, datetime

`tesp_support.api.metrics_api.`**`get_system_energy_loss`**(*energy_sold_df*, *sold_col_id*, *energy_purchased_df*, *purchased_col_id*, *start_date_time*, *duration*)

Function calculates the energy losses inclusive of transmission and distribution losses

Metric defined in document VM_Total System Losses.docx

Parameters

- **energy_sold_df** (*dataframe*) – time series dataframe containing the 5-minute sold energy data
- **sold_col_id** (*str*) – name of the dataframe column where the sold data is located
- **energy_purchased_df** (*dataframe*) – time series dataframe containing the 5-minute purchased energy data
- **purchased_col_id** (*str*) – name of the dataframe column where the purchased data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated hourly energy loss

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_system_energy_losses`**(*feeder_generation_df*, *gen_col_id*, *feeder_load_df*, *feeder_col_id*, *start_date_time*, *duration*)

Function calculates the total energy loss at a feeder

Metric is defined in document VM_System Energy Losses.docx

Parameters

- **feeder_generation_df** (*dataframe*) – time series dataframe containing the 5-minute total feeder generation data
- **gen_col_id** (*str*) – name of the dataframe column where the generation data is located
- **feeder_load_df** (*dataframe*) – time series dataframe containing the 5-minute total feeder load data
- **feeder_col_id** (*str*) – name of the dataframe column where the load data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated hourly energy losses

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_total_pv_reactive_power`**(*time_series*, *pv_col_id*, *start_date_time*, *duration*)

Function calculates the hourly total system reactive power generated from PV

Metric defined in document VM_Total PV Reactive Power.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the 5-minute power data used in the calculations
- **pv_col_id** (*str*) – name of the dataframe column where the power data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated hourly total reactive power values

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_total_pv_real_power`**(*time_series*, *pv_col_id*, *start_date_time*, *duration*)

Function calculates the hourly total system reactive power generated from PV

Metric is defined in document VM_Total PV Real Power.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the 5-minute power data used in the calculations
- **pv_col_id** (*str*) – name of the dataframe column where the power data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Return type
dataframe

`tesp_support.api.metrics_api.get_total_wind_reactive_power(time_series, power_col_id, start_date_time, duration)`

Function calculates the hourly total system reactive power generated from Wind

Metric defined in document VM_Total Wind Reactive Power.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the 5-minute wind reactive power data
- **power_col_id** (*str*) – name of the dataframe column where the wind data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the hourly wind power results

Return type
dataframe

`tesp_support.api.metrics_api.get_total_wind_real_power(time_series, power_col_id, start_date_time, duration)`

Function calculates the hourly total system real power generated from wind

Metric defined in document VM_Total Wind Real Power.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the 5-minute wind power data
- **power_col_id** (*str*) – name of the dataframe column where the wind data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the hourly total wind data results

Return type
dataframe

`tesp_support.api.metrics_api.get_transmission_over_voltage(time_series, voltage_col_id, compare_val, start_date_time, duration)`

Function calculates the maximum over-voltage violations at the transmission node

Metric defined in document VM_Transmission Over-Voltage Violation.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the 3-phase transmission node voltage
- **voltage_col_id** (*str*) – name of the dataframe column containing the voltage data
- **compare_val** (*float*) – threshold value to compare the data against

- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the hourly transmission over voltage results

Return type
dataframe

`tesp_support.api.metrics_api.get_transmission_under_voltage(time_series, voltage_col_id, compare_val, start_date_time, duration)`

Function calculates the maximum under-voltage violations at the transmission node

Metric defined in document VM_Transmission Under-Voltage Violation.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the 5-minute voltage data
- **voltage_col_id** (*str*) – name of the column where the voltage data is located
- **compare_val** (*float*) – threshold value to compare the voltage data against
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated hourly under voltage results

Return type
dataframe

`tesp_support.api.metrics_api.get_transmission_voltage_magnitude(time_series, column_id, start_date, duration)`

Function calculates the hourly min, max, and avg values from the five-minute data contained in the time_series dataframe

Metric defined in document VM_Transmission Voltage Magnitude.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the five-minute data
- **column_id** (*str*) – the name of the dataframe column with contains the transmission voltage data
- **start_date** (*datetime*) – the starting date and time when the calculations should take place
- **duration** (*int*) – the duration in hours to calculate the ending date and time when the calculations should take place

Returns
the calculated hourly min, max, and average values in a time series dataframe

Return type
dataframe

`tesp_support.api.metrics_api.get_under_voltage_count(time_series, val_col_id, minimum_value)`

Function calculates number of under-voltage violations during the year

Metric defined in VM_Count of Transmission Under-Voltage Violation Events.docx

Parameters

- **time_series** (*dataframe*) – dataframe containing a time series of transmission values
- **val_col_id** (*str*) – The id of the dataframe column where the transmission values are located
- **minimum_value** (*float*) – The value that is to be used to compare transmission values against

Returns
time series dataframe containing a column with the under voltage counts

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_unserved_electric_load`**(*supply_df*, *supply_col_id*, *demand_df*, *demand_col_id*, *start_date_time*)

Function calculates the demand that was not met by supply during the course of 8760 hours

Metric defined in document VM_Unserved Electric Load.docx

Parameters

- **supply_df** (*dataframe*) – hourly supply data per year
- **supply_col_id** (*str*) – name of the dataframe column where the supply data is located
- **demand_df** (*dataframe*) – hourly demand data per year
- **demand_col_id** (*str*) – name of the dataframe column where the demand data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start

Returns
time series dataframe containing the calculated unserved load data

Return type
dataframe

`tesp_support.api.metrics_api.`**`get_valuation`**(*time_series*, *start_date*, *column_index*)

Function calculates the average power generated by solar photovoltaic (PV) power generators aggregated by hour of day.

Metric defined in document VM_Distribution of PV real power generation by hour.docx

Parameters

- **time_series** (*dataframe*) – dataframe containing the timeseries data to be used to calculate the valuations
- **start_date** (*datetime*) – the starting date when the calculations will be started
- **column_index** (*str*) – the dataframe column id that is used to identify the location of the values in the dataframe

Returns

function returns a tuple containing the dataframe containing the valuation
values as a time series, a float representing the 14th percentile of the values, and a float representing the 86th percentile of the values

Return type
dataframe, float, float

`tesp_support.api.metrics_api.get_wind_energy_production(time_series, prod_col_id, start_date_time, duration)`

Function calculates the amount of energy produced by wind at a feeder

Metric defined in document VM_Wind Energy Production.docx

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the 5-minute wind energy production data
- **prod_col_id** (*str*) – name of the column where the wind energy production data is located
- **start_date_time** (*str*) – the starting date and time when the calculation should start
- **duration** (*int*) – the length of time in hours which the calculations should be executed

Returns
time series dataframe containing the calculated hourly total wind energy results

Return type
dataframe

`tesp_support.api.metrics_api.synch_series(time_series, synch_interval, interval_unit)`

Function synchronizes all the time series data frames in the time_series list, so they all have the same start and ending times and the same number of times based upon a shared sampling interval

Parameters

- **time_series** (*list<dataframe>*) – time series dataframe
- **synch_interval** (*int*) – the size of the time intervals to be used in the time series
- **interval_unit** (*str*) – the unit of the time interval the time series is to be sampled "T", "H", "S"

Returns
time series dataframe containing the resampled data of the original

Return type
list<dataframe>

`tesp_support.api.metrics_api.synch_series_lengths(time_series)`

Function clips each of the time series in the time_series list so that each time series data frame has the same start and ending times

Parameters
time_series (*list<dataframe>*) – List containing a set of pandas dataframes each representing a time series

Returns
a list containing the clipped time series data frames

Return type
list<dataframe>

`tesp_support.api.metrics_api.synch_time_series(series_list, synch_interval, interval_unit)`

Function resamples the time steps of the dataframes contained in the input data frame list to match the time intervals specified in the inputs

Parameters

- **series_list** (*list<dataframe>*) – List containing a set of pandas dataframes each representing a time series

- **synch_interval** (*int*) – the size of the time step which should be used to resample the dataframe
- **interval_unit** (*str*) – the measurement unit of the interval to be sampled. The options for this function include the following options "nanoseconds", "seconds", "minutes", "hours", "days", "months", "years"

Returns
pandas dataframe time series containing the resampled columns of data

Return type
list<dataframe>

tesp_support.api.metrics_base_api module

`tesp_support.api.metrics_base_api.`**`adjust_date_time`**(*start_date*, *offset_type*, *offset_val*)

Function returns a date time object that is calculated by adding the offset_val to the entered start date

Parameters

- **start_date** – (datetime) the start date time
- **offset_type** (*str*) – defines what interval of time is to be used. The following identifiers can be used "years", "months", "days", "hours", "minutes", "seconds", "nanoseconds"
- **offset_val** (*int*) – the number of time intervals that are to be added to start_time

Returns
the modified date time

Return type
dataframe

`tesp_support.api.metrics_base_api.`**`check_dataframe_synchronization`**(*data_frame_1*, *data_frame_2*)

Function checks that two time series dataframes are synchronized by comparing size, starting time, and ending time of the data sets. If they are synchronized, the returns "Synchronized". If they are not, then the function will return an error message dependent upon what test failed.

Parameters

- **data_frame_1** (*dataframe*) – time series dataframe
- **data_frame_2** (*dataframe*) – time series dataframe

Returns
a "Synchronized" message if the two dataframes are synchronized, if not, the function returns an error message

Return type
str

`tesp_support.api.metrics_base_api.`**`check_for_5_minute_data`**(*time_series*)

Function checks if the data in the time series is 5-minute intervals

Parameters
time_series (*dataframe*) – time series dataframe containing 5-minute data records

Returns
True if the data in the dataframe is 5-minute and False if it is not

Return type
bool

`tesp_support.api.metrics_base_api.check_for_full_year_data(time_series)`

Function checks if the input time series dataframe contains a full year's worth of data

Parameters
time_series (*dataframe*) – time series dataframe containing a year's worth of data

Returns
True if the time series contains a full year's worth of data

Return type
bool

`tesp_support.api.metrics_base_api.check_for_hourly_data(time_series)`

Function checks if the data in the time series is hourly

Parameters
time_series (*dataframe*) – time series dataframe containing hourly data

Returns
True if the data in the dataframe is hourly and False if it is not hourly

Return type
bool

`tesp_support.api.metrics_base_api.create_testing_dataframe(start_date, end_date, col_names, time_interval)`

Function creates a testing dataframe containing random values

Parameters

- **start_date** (*str*) – the starting date of the time series
- **end_date** (*str*) – then ending date of the time series
- **col_names** (*list<string>*) – the names to be used as the column headers in the resultant dataframe
- **time_interval** (*int*) – frequency of time intervals. These designations are the same as the designations used to define a pandas date_range e.g. "T", "5T", "H", "12H",…

Returns
time series dataframe containing random data values over the course of the defined time range

Return type
dataframe

`tesp_support.api.metrics_base_api.get_accuracy_ratio(input_df, actual_index, simulated_index)`

Function calculates the ratio of simulated data to actual data

Parameters

- **input_df** (*dataframe*) – time series dataframe containing data columns for actual and simulated values
- **actual_index** (*str*) – column id where the actual data is located
- **simulated_index** (*str*) – column id where the simulated data is located

Returns
time series dataframe containing the calculated ratio values

Return type
dataframe

`tesp_support.api.metrics_base_api.get_avg_column_value(time_series, val_index)`

Function calculates the mean of the values in a data column of a dataframe

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be averaged
- **val_index** (*str*) – name of the column that contains the data to be averaged

Returns
calculated average value for the column identified in the function arguments

Return type
float

`tesp_support.api.metrics_base_api.get_avg_data_value(time_series, column_id)`

Function calculates the average of a column in the time series dataframe and returns the average value of the column

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be averaged
- **column_id** (*str*) – name of the data column for which the average is to be calculated

Returns
calculated average for the identified dataframe column

Return type
float

`tesp_support.api.metrics_base_api.get_column_total_value(time_series, column_id)`

Function returns the sum of the values in a dataframe column

Parameters

- **time_series** (*dataframe*) – the time series dataframe which contains the data to be summed
- **column_id** (*str*) – name of the column containing the values to be summed

Returns
the sum of the values contained in the identified dataframe column

Return type
float

`tesp_support.api.metrics_base_api.get_max_column_value(time_series, val_index)`

Function searches a designated column in the time series dataframe and returns the maximum value found in the column

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be searched for a maximum value
- **val_index** (*str*) – name of the column where the data is located to calculate the maximum value

Returns
the maximum data value found in the designated column

Return type
float

`tesp_support.api.metrics_base_api.get_min_column_value(time_series, val_index)`

Function searches a designated column in the time series dataframe and returns the minimum value found in the column

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be searched for a minimum value
- **val_index** (*str*) – name of the column where the data is located to calculate the minimum value

Returns
the minimum data value found in the designated column

Return type
dataframe

`tesp_support.api.metrics_base_api.get_node_data(time_series, node_id, id_column_name)`

Function queries the time series dataframe for the data set identified by the entered node id

Parameters

- **time_series** (*dataframe*) – time series dataframe contains the node data to be queried
- **node_id** (*str*) – the id of the node that is to be used in the query
- **id_column_name** (*str*) – name of the time series dataframe column that contains the node ids to be queried against

Returns
time series dataframe containing data specific to a single node id

Return type
dataframe

`tesp_support.api.metrics_base_api.get_node_ids(time_series, id_column_name)`

Function queries a list of unique values from a time series dataframe based upon a column id entered by the user

Parameters

- **time_series** (*dataframe*) – time series dataframe that contains the values to be queried
- **id_column_name** (*str*) – name of the dataframe column where the values are located

Returns
list object containing the unique values found in the time series dataframe

Return type
list

`tesp_support.api.metrics_base_api.get_time_series_average(time_series, start_date, duration)`

Function calculates the average of each data column in the dataframe

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be averaged
- **start_date** (*str*) – the starting date and time that should be used in the calculation of the averages
- **duration** (*int*) – the duration in hours that the averages should be calculated

Returns
dataframe containing the average value for each column in the input dataframe

Return type
dataframe

`tesp_support.api.metrics_base_api.`**`get_time_series_difference_values`**(*time_series*, *column_id*, *time_series2*, *column_id2*)

Function calculates the difference between data in a column of a dataframe with the data in a column of a second dataframe

Parameters

- **time_series** (*dataframe*) – time series dataframe containing a data set to be used in the calculation
- **column_id** (*str*) – name of the column where the data to be used is located
- **time_series2** (*dataframe*) – time series dataframe containing a data set to be used in the calculation
- **column_id2** (*str*) – name of the column where the data to be used is located

Returns
the total value difference calculated as time_series2 - time_series1

Return type
float

`tesp_support.api.metrics_base_api.`**`get_time_series_max_value_over`**(*time_series*, *column_id*, *compare_value*)

Function calculates the maximum value out of the number of values in a dataframe column that are greater than a comparison value

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be compared
- **column_id** (*str*) – the name of the column in the dataframe where the data is located
- **compare_value** (*str*) – the value the data is to be compared with

Returns
the maximum of the values that are greater than the compare value

Return type
int

`tesp_support.api.metrics_base_api.`**`get_time_series_max_value_under`**(*time_series*, *column_id*, *compare_value*)

Function calculates the maximum value out of the number of values in a dataframe column that are less than a comparison value

Parameters

- **time_series** (*dataframe*) – time series dataframe containing the data to be compared
- **column_id** (*str*) – the name of the column in the dataframe where the data is located
- **compare_value** (*str*) – the value the data is to be compared with

Returns
the maximum of the values that are less than the compare value

Return type
int

tesp_support.api.metrics_collector module

Utility functions for metrics collection within tesp_support, able to write to JSON and HDF5

class tesp_support.api.metrics_collector.**MetricsCollector**(*start_time='1970-01-01 00:00:00'*)

Bases: object

Metrics collector base class that handles collecting and writing data to disk (.json).

start_time

the start time of the simulation

Type
pd.Timestamp

metrics_stores

list of MetricsStores holding/growing data

Type
list

classmethod factory(*start_time='1970-01-01 00:00:00'*, *write_hdf5=False*)

Parameters

- **start_time** (*str*) – start time of simulation in datetime string format
- **write_hdf5** (*bool*) – flag to determine if we write to .h5 (if True) or .json (if False; defaults to this)

Returns
MetricsCollectorHDF or Base instance, depending on write_hdf5 flag

Return type
MetricsCollector

finalize_writing()

register_metrics_store(*metrics_store*)

Parameters
metrics_store (MetricsStore) – A store to be appended to our ongoing list

write_metrics()

Write all known metrics to disk (.json) and reset data within each metric.

class tesp_support.api.metrics_collector.**MetricsCollectorHDF**(*start_time='1970-01-01 00:00:00'*)

Bases: *MetricsCollector*

finalize_writing()

write_metrics()

Write all known metrics to disk (.h5).

class tesp_support.api.metrics_collector.**MetricsStore**(*name_units_pairs*, *file_string*, *collector*)

Bases: object

This stores our metrics in appropriately sized tables, geared towards being ready to write to hdf5 (so writing to json might take longer than if we kept things ready to write to json).

time_uid_pairs

an ongoing list of (time, uid) pairs incoming with data

Type
list

index_to_shapes

shapes of incoming column's units

Type
list

file_string

the file path (barring extension) which will be appended with "_metrics.{h5, json}"

Type
str

collector

a common store for these metrics, to ease writing out all metrics/tables

Type
MetricsCollector

append_data(*time*, *uid*, **args*)

Appends a single (time, uid) pair's metrics to appropriate tables (depends on shape of each arg)

time (str or int): time in seconds after start of simulation uid (str or int or ?): unique identifier of an object (e.g. a name) args (list): an list of length/order equal to name_units_pairs seen when constructing this store

clear()

class tesp_support.api.metrics_collector.**MetricsTable**(*columns*, *units*)

Bases: object

append_data(*data*)

clear()

to_frame(*times*, *uids*, *shape*, *filename=''*)

tesp_support.api.metrics_collector.**deepish_copy**(*obj*)

Faster approach to deepcopy, for an object of the simple python types. :param obj: original object to copy

Returns
copy of obj

Return type
object

tesp_support.api.metrics_collector.**finalize_hdf**(*metrics_store*)

tesp_support.api.metrics_collector.**to_hdf**(*metrics_store*, *start_time*, *num_writes_counter*)

This function writes the metric data to HDF5 files (and clears the data)

Parameters

- **metrics_store** (MetricsStore) – a store containing metrics tables to dump to file
- **start_time** (*pd.Timestamp*) – start time of simulation times

- **num_writes_counter** (*int*) – interval counter

`tesp_support.api.metrics_collector.to_json(metrics_store, start_time)`

This function writes the metric data to JSON files (and clears the data) :param metrics_store: a store containing metrics tables to dump to file :type metrics_store: MetricsStore :param start_time: start time of simulation times :type start_time: pd.Timestamp

tesp_support.api.model_GLM module

tesp_support.api.modify_GLM module

tesp_support.api.parse_helpers module

`tesp_support.api.parse_helpers.parse_helic_input(arg)`

Helper function to find the magnitude of a possibly complex number from Helics as a string

Parameters
arg (*str*) – The Helics value

Returns
the parsed number, or 0 if parsing fails

Return type
float

`tesp_support.api.parse_helpers.parse_kva(arg)`

Parse the kVA magnitude from GridLAB-D P+jQ volt-amperes in rectangular form

Parameters
arg (*str*) – the GridLAB-D P+jQ value

Returns
the parsed kva value

Return type
float

`tesp_support.api.parse_helpers.parse_kva_old(arg)`

Parse the kVA magnitude from GridLAB-D P+jQ volt-amperes in rectangular form

Parameters
arg (*str*) – the GridLAB-D P+jQ value

Returns
the parsed kva value

Return type
float

`tesp_support.api.parse_helpers.parse_kw(arg)`

Parse the kilowatt load of a possibly complex number from FNCS

Parameters
arg (*str*) – the FNCS string value

Returns
the parsed number in kW, or 0 if parsing fails

Return type
float

`tesp_support.api.parse_helpers.parse_magnitude(arg)`

Parse the magnitude of a possibly complex number from FNCS

Parameters
arg (*str*) – the FNCS string value

Returns
the parsed number, or 0 if parsing fails

Return type
float

`tesp_support.api.parse_helpers.parse_magnitude_1(arg)`

Parse the magnitude of a possibly complex number from FNCS

Parameters
arg (*str*) – the FNCS string value

Returns
the parsed number, or 0 if parsing fails

Return type
float

`tesp_support.api.parse_helpers.parse_magnitude_2(arg)`

Helper function to find the magnitude of a possibly complex number from FNCS

Parameters
arg (*str*) – The FNCS value

Returns
the parsed number, or 0 if parsing fails

Return type
float

`tesp_support.api.parse_helpers.parse_mva(arg)`

Helper function to parse P+jQ from a FNCS value

Parameters
arg (*str*) – FNCS value in rectangular format

Returns
P [MW] and Q [MVAR]

Return type
float, float

`tesp_support.api.parse_helpers.parse_number(arg)`

Parse floating-point number from a FNCS message; must not have leading sign or exponential notation

Parameters
arg (*str*) – the FNCS string value

Returns
the parsed number

Return type
float

tesp_support.api.player module

`tesp_support.api.player.load_player_loop(casename, keyName)`

`tesp_support.api.player.make_player(data_path)`

Parameters

- **name** (*str*) – name of the player
- **prefix** (*str*) – the prefix for the file *prefix_player.json* name and the federate *prefix_player* name
- **vals_rows** (*int*) – number of rows in the file
- **power_factor** (*float*) – power factor a load file, 0 for generator file
- **dt_load_collector** (*int*) – interval between rows
- **date_time_str** (*str*) – date and time "YYYY-MM-DD HH:MM:SS"
- **data_path** (*str*) – the data file to be published
- **output** (*bool*) – If true the player outputs data given at the dt_load_collector interval
- **output_hist** (*bool*) – If true the player outputs data history
- **load** (*bool*) – If true this a load player, else a generator player

Return type
dict

tesp_support.api.process_eplus module

Functions to plot data from the EnergyPlus agent

Public Functions:

process_eplus
Reads the data and metadata, then makes the plots.

`tesp_support.api.process_eplus.plot_eplus(diction, title=None, save_file=None, save_only=False)`

`tesp_support.api.process_eplus.process_eplus(name_root, title=None, save_file=None, save_only=False)`

Plots the min and max line-neutral voltages for every billing meter

This function reads *eplus_[name_root]_metrics.json* for both metadata and data. This must exist in the current working directory. One graph is generated with 3 subplots:

1. Cooling system setpoint, actual temperature and the difference between them.
2. Heating system setpoint, actual temperature and the difference between them.
3. Price that the building controller responded to.

Parameters

- **name_root** (*str*) – name of the TESP case, not necessarily the same as the EnergyPlus case, without the extension
- **title** (*str*) – supertitle for the page of plots.

- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.

`tesp_support.api.process_eplus.`**`read_eplus_metrics`**`(`*path*, *name_root*, *quiet=False*`)`

tesp_support.api.process_gld module

Functions to plot data from GridLAB-D

Public Functions:

process_gld
Reads the data and metadata, then makes the plots.

`tesp_support.api.process_gld.`**`plot_gld`**`(`*diction*, *save_file=None*, *save_only=False*`)`

`tesp_support.api.process_gld.`**`process_gld`**`(`*name_root*, *diction_name=''*, *save_file=None*, *save_only=False*`)`

Plots a summary/sample of power, air temperature and voltage

This function reads *substation_[name_root]_metrics.json*, *billing_meter_[name_root]_metrics.json* and *house_[name_root]_metrics.json* for the data; it reads *[name_root]_glm_dict.json* for the metadata. These must all exist in the current working directory. Makes one graph with 4 subplots:

1. Substation real power and losses
2. Average air temperature over all houses
3. Min/Max line-to-neutral voltage and Min/Max line-to-line voltage at the first billing meter
4. Min, Max and Average air temperature at the first house

Parameters

- **name_root** (*str*) – name of the TESP case, not necessarily the same as the GLM case, without the extension
- **diction_name** (*str*) – metafile name (with json extension) for a different GLM dictionary, if it's not *[name_root]_glm_dict.json*. Defaults to empty.
- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.

`tesp_support.api.process_gld.`**`read_gld_metrics`**`(`*path*, *name_root*, *diction_name=''*`)`

tesp_support.api.process_houses module

Functions to plot house data from GridLAB-D

Public Functions:

process_houses
Reads the data and metadata, then makes the plot.

`tesp_support.api.process_houses.plot_houses(diction, save_file=None, save_only=False)`

`tesp_support.api.process_houses.process_houses(name_root, diction_name='', save_file=None, save_only=True)`

Plots the temperature and HVAC power for every house

This function reads *substation_[name_root]_metrics.json* and *house_[name_root]_metrics.json* for the data; it reads *[name_root]_glm_dict.json* for the metadata. These must all exist in the current working directory. Makes one graph with 2 subplots:

1. Average air temperature at every house
2. Average HVAC power at every house

Parameters

- **name_root** (*str*) – name of the TESP case, not necessarily the same as the GLM case, without the extension
- **diction_name** (*str*) – metafile name (with json extension) for a different GLM dictionary, if it's not *[name_root]_glm_dict.json*. Defaults to empty.
- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.

`tesp_support.api.process_houses.read_houses_metrics(path, name_root, diction_name='')`

tesp_support.api.process_inv module

Functions to plot inverter and volt-var data from GridLAB-D, for NIST TE Challenge 2

Public Functions:

process_inv
Reads the data and metadata, then makes the plots.

`tesp_support.api.process_inv.plot_inv(diction, save_file=None, save_only=False)`

`tesp_support.api.process_inv.process_inv(name_root, diction_name='', title=None, save_file=None, save_only=False)`

Plots inverter and volt-var data for the NIST TE Challenge 2 / IEEE 8500 examples

This function reads *substation_[name_root]_metrics.json*, *billing_meter_[name_root]_metrics.json*, *capacitor_[name_root]_metrics.json*, *regulator_[name_root]_metrics.json*, *house_[name_root]_metrics.json* and *inverter_[name_root]_metrics.json* for the data; it reads *[name_root]_glm_dict.json* for the metadata. If possible, it reads *precool_[name_root]_metrics.json* for temperature deviation. These must all exist in the current working directory. One graph is generated with 10 subplots:

1. Average P and Q over all inverters
2. Min, Max and Average line-neutral voltage over all billing meters
3. Average air temperature over all houses
4. Average temperature deviations from the setpoint over all houses
5. Total of ANSI C84 A and B range violation counts, summing over all billing meters
6. Total of ANSI C84 A and B range violation durations, summing over all billing meters
7. Substation total power, losses, house power, house HVAC power and house waterheater power
8. The accumulated bill, summed over all billing meters
9. The accumulated capacitor switching counts for each of 4 capacitor banks, if found, as in the IEEE 8500 case
10. The accumulated regulator counts for each of 4 voltage regulators, if found, as in the IEEE 8500 case

Parameters

- **name_root** (*str*) – name of the TESP case, not necessarily the same as the GLM case, without the extension
- **diction_name** (*str*) – metafile name (with json extension) for a different GLM dictionary, if it's not *[name_root]_glm_dict.json*. Defaults to empty.
- **title** (*str*) –
- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.

`tesp_support.api.process_inv.`**`read_inv_metrics`**(*path*, *name_root*, *diction_name=''*)

tesp_support.api.process_pypower module

Functions to plot bus and generator data from PYPOWER

Public Functions:

process_pypower
Reads the data and metadata, then makes the plots.

`tesp_support.api.process_pypower.`**`plot_pypower`**(*diction*, *title=None*, *save_file=None*, *save_only=False*)

`tesp_support.api.process_pypower.`**`process_pypower`**(*name_root*, *title=None*, *save_file=None*, *save_only=True*)

Plots bus and generator quantities for the 9-bus system used in te30 or sgip1 examples

This function reads *bus_[name_root]_metrics.json* and *gen_[name_root]_metrics.json* for the data, and *[name_root]_m_dict.json* for the metadata. These must all exist in the current working directory. One graph is generated with 8 subplots:

1. Bus P and Q demands, at the single GridLAB-D connection
2. Bus P and Q locational marginal prices (LMP), at the single GridLAB-D connection
3. Bus Vmin, Vmax and Vavg, at the single GridLAB-D connection

4. All 4 generator prices
5. Generator 1 P and Q output
6. Generator 2 P and Q output
7. Generator 3 P and Q output
8. Generator 4 P and Q output

Parameters

- **name_root** (*str*) – file name of the TESP case, not necessarily the same as the PYPOWER case, w/out the JSON extension
- **title** (*str*) – supertitle for the page of plots.
- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.

`tesp_support.api.process_pypower.read_pypower_metrics(path, name_root)`

tesp_support.api.process_voltages module

Functions to plot all billing meter voltages from GridLAB-D

Public Functions:

process_voltages
Reads the data and metadata, then makes the plot.

`tesp_support.api.process_voltages.plot_voltages(diction, save_file=None, save_only=False)`

`tesp_support.api.process_voltages.process_voltages(name_root, diction_name='', save_file=None, save_only=True)`

Plots the min and max line-neutral voltages for every billing meter

This function reads *substation_[name_root]_metrics.json* and *billing_meter_[name_root]_metrics.json* for the voltage data, and *[name_root]_glm_dict.json* for the meter names. These must all exist in the current working directory. One graph is generated with 2 subplots:

1. The Min line-to-neutral voltage at each billing meter
2. The Max line-to-neutral voltage at each billing meter

Parameters

- **name_root** (*str*) – name of the TESP case, not necessarily the same as the GLM case, without the extension
- **diction_name** (*str*) – metafile name (with json extension) for a different GLM dictionary, if it's not *[name_root]_glm_dict.json*. Defaults to empty.
- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.

tesp_support.api.process_voltages.**read_voltages_metrics**(*path*, *name_root*, *diction_name=''*)

tesp_support.api.schedule_client module

class tesp_support.api.schedule_client.**DataClient**(*port*)

Bases: object

tesp_support.api.schedule_server module

tesp_support.api.store module

Path and Data functions for use within tesp_support, including new agents.

class tesp_support.api.store.**Directory**(*file*, *description=None*)

Bases: object

get_includeDirs()

get_includeFiles(*path*)

set_includeDir(*path*, *recurse=False*)

set_includeFile(*path*, *mask*)

toJSON()

zip(*zipfile*)

class tesp_support.api.store.**Schema**(*file*, *name=None*, *description=None*)

Bases: object

get_columns(*table*, *skip_rows=0*)

get_date(*table*)

get_series_data(*table*, *start*, *end*, *usecols=None*, *index_col=None*)

get_tables()

set_date_bycol(*table*, *name*)

set_date_byrow(*table*, *start*, *interval*)

toJSON()

class tesp_support.api.store.**Store**(*file*)

Bases: object

add_directory(*directory*)

add_file(*path*, *name=''*, *description=''*)

add_path(*path*, *description=''*)

add_schema(*scheme*)

del_directory(*name*)

del_schema(*name*)

get_directory(*name*)

get_schema(*name=None*)

read()

write()

zip()

tesp_support.api.store.**test_hdf5**()

tesp_support.api.store.**unzip**(*file*, *path*)

This function unzip take name add .zip and unzip the file to specified path. Then finds the store json in that path and fixes the relative path for the stores work

tesp_support.api.substation module

Manages the simple_auction and hvac agents for the te30 and sgip1 examples

Public Functions:

substation_loop
initializes and runs the agents

Todo:

- Getting an overflow error when killing process - investigate whether that happens if simulation runs to completion
- Allow changes in the starting date and time; now it's always midnight on July 1, 2013
- Allow multiple markets per substation, e.g., 5-minute and day-ahead for the DSO+T study

---

tesp_support.api.test_runner module

Auto test runner for TESP run* cases Runs a test case based on pre-existing shell script file.

If FNCS or HELICS broker exist the test waits for the broker process to finish before function returns.

This code has limited functionality as the 'run*' scripts for the examples are written in a very specified way.

tesp_support.api.test_runner.**block_test**(*call*)

tesp_support.api.test_runner.**docker_line**(*line*, *local_vars*)

tesp_support.api.test_runner.**exec_test**(*file_name*, *case_name=None*)

tesp_support.api.test_runner.**init_tests**()

tesp_support.api.test_runner.**process_line**(*line*, *local_vars*)

tesp_support.api.test_runner.**report_tests**()

`tesp_support.api.test_runner.run_docker_test(file_name, case_name=None)`

`tesp_support.api.test_runner.run_test(file_name, case_name=None)`

`tesp_support.api.test_runner.services(name, image, env, cnt, outfile, depends=None)`

`tesp_support.api.test_runner.start_test(case_name=None)`

tesp_support.api.time_helpers module

Utility time functions for use within tesp_support, including new agents.

`tesp_support.api.time_helpers.add_hhmm_secs(hhmm, secs)`

Add hhmm time + seconds duration

Parameters

- **hhmm** (*float*) – HHMM
- **secs** (*int*) – seconds

Returns
hhmm+secs in hhmm format

`tesp_support.api.time_helpers.get_dist(mean, var)`

Get a random number from a distribution given mean and %variability

Parameters

- **mean** (*float*) – mean of distribution
- **var** (*float*) – % variability

Returns
one random entry from distribution

Return type
float

`tesp_support.api.time_helpers.get_duration(arrival, leave)`

Convert arrival and leaving time to duration

Parameters

- **arrival** (*float*) – in HHMM format
- **leave** (*float*) – in HHMM format

Returns
duration in seconds

Return type
int

`tesp_support.api.time_helpers.get_hhmm_from_secs(time)`

Convert seconds to HHMM

Parameters
time (*int*) – seconds

Returns
HHMM

Return type
int

`tesp_support.api.time_helpers.get_secs_from_hhmm(time)`

Convert HHMM to seconds

Parameters
time (*float*) – HHMM

Returns
seconds

Return type
int

`tesp_support.api.time_helpers.is_hhmm_valid(time)`

Check if HHMM is a valid number

Parameters
time (*float*) – HHMM format

Returns
true if valid or false if not

Return type
bool

`tesp_support.api.time_helpers.subtract_hhmm_secs(hhmm, secs)`

Subtract hhmm time - secs duration

Parameters

- **hhmm** (*float*) – HHMM format time
- **secs** (*int*) – seconds

Returns
arrival time in HHMM

tesp_support.api.tso_PYPOWER module

PYPOWER solutions under control of HELICS for te30 and dsot, sgip1 examples

Public Functions:

pypower_loop
Initializes and runs the simulation.

tesp_support.api.tso_helpers module

Helpers for PYPOWER, PSST, MOST solutions

Public Functions:
print_matrix, print_keyed_matrix, load_json_case, print_mod_load, summarize_opf, make_dictionary, dist_slack

`tesp_support.api.tso_helpers.dist_slack(mpc, prev_load)`

Parameters

- **mpc** (*dict*) – PYPOWER case structure

- **prev_load** (*float*) – previous load

Returns
generator loads

Return type
array

`tesp_support.api.tso_helpers.load_json_case(file_name)`

Helper function to load PYPOWER case from a JSON file

Parameters
file_name (*str*) – the JSON file name to open

Returns
the loaded PYPOWER case structure

Return type
dict

`tesp_support.api.tso_helpers.make_dictionary(mpc)`

Helper function to write the JSON model dictionary metafile for post-processing

Additions to DSO, and Pnom==>Pmin for generators are there

Parameters
mpc (*dict*) – PYPOWER case file structure based on matpower

`tesp_support.api.tso_helpers.print_keyed_matrix(lbl, D, fmt='{:8.4f}')`

`tesp_support.api.tso_helpers.print_m_case(ppc, ppc_case)`

`tesp_support.api.tso_helpers.print_matrix(lbl, A, fmt='{:8.4f}')`

`tesp_support.api.tso_helpers.print_mod_load(bus, dso, model_load, msg, ts)`

`tesp_support.api.tso_helpers.summarize_opf(mpc)`

Helper function to print optimal power flow solution (debugging)

Parameters
mpc (*dict*) – solved using PYPOWER case structure

tesp_support.api.tso_psst module

`tesp_support.api.tso_psst.make_generator_plants(ppc, renewables)`

tesp_support.consensus package

Transactive Energy Simulation Platform (TESP) Contains the python files for the Consensus analysis

Submodules

tesp_support.consensus.case_merge module

Combines GridLAB-D and agent files to run a multi-feeder TESP simulation

Public Functions:

merge_glm
combines GridLAB-D input files

merge_glm_dict
combines GridLAB-D metadata files

merge_agent_dict
combines the substation agent configuration files

merge_substation_yaml
combines the substation agent FNCS publish/subscribe files

merge_fncs_config
combines GridLAB-D FNCS publish/subscribe files

`tesp_support.consensus.case_merge.`**`merge_agent_dict`**(*target*, *sources*)

Combines the substation agent configuration files into "target". The source files must already exist.

Parameters

- **target** (*str*) – the path to the target JSON file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.consensus.case_merge.`**`merge_fncs_config`**(*target*, *dso*, *sources*)

Combines GridLAB-D input files into "target". The source feeders must already exist.

Parameters

- **target** (*str*) – the path to the target TXT file, including the name of the file
- **dso** (*str*) – dso id
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.consensus.case_merge.`**`merge_glm`**(*target*, *sources*, *xfmva*)

Combines GridLAB-D input files into "target". The source files must already exist.

Parameters

- **target** (*str*) – the path to the target GLM file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge
- **xfmva** (*int*) –

`tesp_support.consensus.case_merge.`**`merge_glm_dict`**(*target*, *sources*, *xfmva*)

Combines GridLAB-D metadata files into "target". The source files must already exist.

The output JSON won't have a top-level base_feeder attribute. Instead, the base_feeder from each source file will become a feeder key in the output JSON feeders dictionary, and then every child object on that feeder will have its feeder_id, originally network_node, changed to match the base_feeder.

Parameters

- **target** (*str*) – the path to the target JSON file, including the name of the file

- **sources** (*list*) – list of feeder names in the target directory to merge
- **xfmva** (*int*) –

`tesp_support.consensus.case_merge.merge_substation_yaml(target, sources)`

Combines GridLAB-D input files into "target". The source files must already exist.

Parameters

- **target** (*str*) – the path to the target YAML file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge

tesp_support.consensus.dg_agent module

Manages the Transactive Control scheme for DSO+T implementation version 1

Public Functions:

substation_loop
initializes and runs the agents

`tesp_support.consensus.dg_agent.destroy_federate(fed)`

`tesp_support.consensus.dg_agent.inner_substation_loop(configfile, metrics_root, with_market)`

Helper function that initializes and runs the DSOT agents

TODO: This needs to be updated Reads configfile. Writes *auction_metrics_root_metrics.json* and *controller_metrics_root_metrics.json* upon completion.

Parameters

- **configfile** (*str*) – fully qualified path to the JSON agent configuration file
- **metrics_root** (*str*) – base name of the case for metrics output
- **with_market** (*bool*) – flag that determines if we run with markets

`tesp_support.consensus.dg_agent.register_federate(json_filename)`

`tesp_support.consensus.dg_agent.substation_loop(configfile, metrics_root, with_market=True)`

Wrapper for *inner_substation_loop*

When *inner_substation_loop* finishes, timing and memory metrics will be printed for non-Windows platforms.

`tesp_support.consensus.dg_agent.worker(arg)`

tesp_support.consensus.dso_agent module

Manages the Transactive Control scheme for DSO+T implementation version 1

Public Functions:

substation_loop
initializes and runs the agents

`tesp_support.consensus.dso_agent.destroy_federate(fed)`

`tesp_support.consensus.dso_agent.inner_substation_loop(configfile, metrics_root, with_market)`

Helper function that initializes and runs the DSOT agents

TODO: This needs to be updated Reads configfile. Writes *auction_metrics_root_metrics.json* and *controller_metrics_root_metrics.json* upon completion.

Parameters

- **configfile** (*str*) – fully qualified path to the JSON agent configuration file
- **metrics_root** (*str*) – base name of the case for metrics output
- **with_market** (*bool*) – flag that determines if we run with markets

`tesp_support.consensus.dso_agent.register_federate(json_filename)`

`tesp_support.consensus.dso_agent.substation_loop(configfile, metrics_root, with_market=True)`

Wrapper for *inner_substation_loop*

When *inner_substation_loop* finishes, timing and memory metrics will be printed for non-Windows platforms.

`tesp_support.consensus.dso_agent.worker(arg)`

tesp_support.consensus.dso_market module

Class that manages the operation of DSO agent

Functionalities include: Aggregate demand bids from different substations; Wholesale no da trial clearing; Conversion between wholesale price and retail price; Generate substation supply curves with and without consideration of the transformer degradation.

class `tesp_support.consensus.dso_market.DSOMarket(dso_dict, key)`

Bases: `object`

This agent manages the DSO operating

Parameters

- **dso_dict** –
- **key** –

name

name of the DSO agent

Type
str

price_cap

the maximum price that is allowed in the market, in $/kWh

Type
float

num_samples

the number of sampling points, describes how precisely the curve is sampled

Type
int

windowLength

length of the planning horizon for the DA market, in hours

Type
int

DSO_Q_max

maximum limit of the DSO load capacity, in kWh

Type
float

transformer_degradation

flag variable, equals to 1 when transformer degradation effect is taken into account

Type
bool

curve_a

array of second order coefficients for the wholesale node curve, indexed by day_of_sim and hour_of_day

Type
array

curve_b

array of first order coefficients of the wholesale node curve, indexed by day_of_sim and hour_of_day

Type
array

curve_c

array of intercepts of the wholesale node curve, indexed by day_of_sim and hour_of_day

Type
array

Pwclear_RT

cleared wholesale price by real-time wholesale node trial clearing, in $/kWh

Type
float

Pwclear_DA

list of cleared wholesale price by day-ahead wholesale node trial clearing, in $/kWh, indexed by hour

Type
list

trial_cleared_quantity_RT

trial cleared quantity by real-time wholesale node trial clearing, in kWh

Type
float

trial_cleared_quantity_DA

trial cleared quantity by day-ahead wholesale node trial clearing, in kWh

Type
list

curve_DSO_RT

aggregated demand curve at DSO level from real-time retail market

Type
Curve

curve_DSO_DA

dictionary of aggregated demand curves at DSO level from day-ahead retail market, indexed by hour

Type
dict

curve_ws_node

dictionary of wholesale node curves, indexed by day_of_sim and hour_of_day

Type
dict

trial_clear_type_RT

trial cleared type of real-time wholesale node trial clearing

Type
int

trial_clear_type_DA

trial cleared type of day-ahead wholesale node trial clearing, indexed by hour

Type
list

hour_of_day

current hour of the day

Type
int

day_of_week

current day of the week

Type
int

customer_count_mix_residential

Residential percentage of the total customer count mix

number_of_gld_homes

Total number of GLD homes for the DSO

clean_bids_DA()

Initialize the day-ahead wholesale node trial clearing

clean_bids_RT()

Initialize the real-time wholesale node trial clearing

curve_aggregator_DSO_DA(*demand_curve_DA*, *Q_max*)

Function used to aggregate the substation-level DA demand curves into a DSO-level DA demand curve

Parameters

- **demand_curve_DA** (*dict*) – a collection of demand curves to be aggregated for day-ahead
- **Q_max** (*float*) – maximum capacity of the substation, in kW

curve_aggregator_DSO_RT(*demand_curve_RT*, *Q_max*)

Function used to aggregate the substation-level RT demand curves into a DSO-level RT demand curve

Parameters

- **demand_curve_RT** (Curve) – demand curve to be aggregated for real-time
- **Q_max** (*float*) – maximum capacity of the substation, in kW

curve_preprocess(*substation_demand_curve*, *Q_max*)

An internal shared function called by curve_aggregator_DSO_RT and curve_aggregator_DSO_DA functions to truncate

the substation demand curve before aggregation as well as convert the retail prices into wholesale prices

Parameters

- **substation_demand_curve** (Curve) – substation demand curve to be preprocessed
- **Q_max** (*float*) – maximum capacity of the substation, in kW

Returns

preprocessed demand curve

Return type

preprocessed_curve (*curve*)

generate_TOC(*costInterval*, *maxPuLoad*, *num_samples*, *TOC_dict*)

Function used to calculate the total owning cost of transformer

Parameters

- **costInterval** (*int*) – interval for calculating the cost, in minutes
- **maxPuLoad** (*float*) – maximum pu loading factor
- **num_samples** (*int*) – number of sampling points
- **TOC_dict** (*dict*) – configuration parameters for transformer

Returns

price axis of unit owning cost, in $/kWh LoadsForPlot (list): quantity axis of unit owing cost, in kWh

Return type

DollarsForPlot (list)

retail_rate(*Pw*)

Function used to convert the wholesale prices into retail prices

Parameters

Pw (*float*) – wholesale price, in $/kWh

Returns

retail price, in $/kWh

Return type

Pr (float)

retail_rate_inverse(*Pr*)

Function used to convert the retail prices into wholesale prices

Parameters
Pr (*float*) – retail price, in $/kWh

Returns
wholesale price, in $/kWh

Return type
Pw (float)

set_Pwclear_DA(*hour_of_day*, *day_of_week*)

Function used to implement the DA trial wholesale node clearing and update the Pwclear_DA value

Parameters

- **hour_of_day** (*int*) – current hour of the day
- **day_of_week** (*int*) – current day of the week

set_Pwclear_RT(*hour_of_day*, *day_of_week*)

Function used to implement the RT trial wholesale node clearing and update the Pwclear_RT value

Parameters

- **hour_of_day** (*int*) – current hour of the day
- **day_of_week** (*int*) – current day of the week

set_cleared_q_da(*val*)

Set the clear quantity for day ahead

Parameters
val (*double*) – lmp for the bus/substation

set_cleared_q_rt(*val*)

Set the clear quantity for real time

Parameters
val (*double*) – lmp for the bus/substation

set_ind_load(*industrial_load*)

Set the industrial load based on provided load by a csv file after base-case, complex number

Parameters
industrial_load (*str*) – industrial load of substation

set_ind_load_da(*industrial_load_da*)

Set the ercot ind load for the next 24-hours provided load by a csv file after base-case, complex number

Parameters
industrial_load_da (*24 x 1 array of double*) – industry load values for the next day ahead are given in MW

set_lmp_da(*val*)

Set the lmp for day ahead

Parameters
val (*array of double*) – lmp for the day ahead

set_lmp_rt(*val*)

Set the lmp for real time

Parameters
val (*double*) – lmp for the bus/substation

set_ref_load(*ref_load*)

Set the reference (ercot) load based on provided load by a csv file after base-case, complex number

Parameters
ref_load (*str*) – total load of substation

set_ref_load_da(*ref_load_da*)

Set the ercot load for the next 24-hours provided load by a csv file after base-case, complex number

Parameters
ref_load_da (*24 x 1 array of double*) – ercot load values for the next day ahead

set_total_load(*total_load*)

Set the residential load based on provided load by GLD, complex number

Parameters
total_load (*str*) – total load of substation

substation_supply_curve_DA(*retail_obj*)

Function used to generate the DA supply curves for each substation

Args:

Variables:
FeederCongPrice (float): feeder congestion price, in $/kWh FeederPkDemandPrice (float): feeder peak demand price, in $/kWh FeederCongCapacity (float): feeder congestion capacity, in kWh FeederPkDemandCapacity (float): feeder peak demand, in kWh Q_max_retail (float): substation limit, in kWh Q_max_DSO (float): can change when the demand bid is higher than the original DSO limit maxPuLoading (float): maximum pu loading factor TOC_dict (dict): configuration parameters for transformer

Returns
a collection of substation supply curves for day-ahead market clearing

Return type
supply_curve_DA (list)

substation_supply_curve_RT(*retail_obj*)

Function used to generate the RT supply curve for each substation

Args:

Variables:
FeederCongPrice (float): feeder congestion price, in $/kWh FeederPkDemandPrice (float): feeder peak demand price, in $/kWh FeederCongCapacity (float): feeder congestion capacity, in kWh FeederPkDemandCapacity (float): feeder peak demand, in kWh Q_max_retail (float): substation limit, in kWh Q_max_DSO (float): can change when the demand bid is higher than the original DSO limit maxPuLoading (float): maximum pu loading factor TOC_dict (dict): configuration parameters for transformer

Returns
substation supply curve for real-time market clearing

Return type
supply_curve_RT (*curve*)

supply_curve(*Prclear*, *FeederCongCapacity*, *FeederPkDemandCapacity*, *num_samples*, *Q_max*, *maxPuLoading*, *TOC_dict*)

An internal shared function called by substation_supply_curve_RT and substation_supply_curve_DA functions to generate the supply curve when considering the transformer degradation

Parameters

- **Prclear** (*float*) – retail price overted from wholesale price obtained from trial wholesale node clearing, in $/kWh
- **FeederCongCapacity** (*float*) – feeder congestion capacity, in kWh
- **FeederPkDemandCapacity** (*float*) – feeder peak demand, in kWh
- **num_samples** (*int*) – number of sampling points
- **Q_max** (*float*) – substation limit, in kWh
- **maxPuLoading** (*float*) – maximum pu loading factor
- **TOC_dict** (*dict*) – configuration parameters for transformer

Variables:
FeederCongPrice (float): feeder congestion price, in $/kWh FeederPkDemandPrice (float): feeder peak demand price, in $/kWh

Returns
quantity sampling of the supply curve, in kWh SupplyPrices (list): prices sampling of the supply curve, in $/kWh

Return type
SupplyQuantities (list)

test_function()

Test function with the only purpose of returning the name of the object

trial_wholesale_clearing(*curve_ws_node*, *curve_DSO*)

An internal shared function called by set_Pwclear_RT and
set_Pwclear_DA functions to implement the trial wholesale node clearing

Parameters

- **curve_ws_node** (Curve) – wholesale node curve
- **curve_DSO** (Curve) – aggregated demand curve at DSO level

Returns
cleared price, in $/kWh cleared_quantity(float): cleared quantity, in kWh trial_clear_type (int): clear type

Return type
Pwclear (float)

update_wholesale_node_curve()

Update the wholesale node curves according to the most updated curve coefficients, may be updated every day

tesp_support.consensus.dso_market.**test**()

tesp_support.consensus.forecasting module

Class responsible for forecasting

Implements the substation level DA price forecast and load forecast

class `tesp_support.consensus.forecasting.`**`Forecasting`**

Bases: `object`

This Class perform the forecast

Parameters

- **TODO** (#) – update inputs
- **TODO** – Load base case run files

TODO

update attributes

add_skew_scalar(*datafr*, *N_skew*, *N_scalar*)

Skew the values with given seconds and multiply by scalar in the whole year dataframe

Args: datafr (DataFrame): dataframe created with the schedule name for a year N_skew (int): number of seconds to skew either (+ or -)

calc_solar_flux(*cpt*, *day_of_yr*, *lat*, *sol_time*, *dnr_i*, *dhr_i*, *vertical_angle*)

calc_solargain(*day_of_yr*, *time*, *dnr*, *dhr*, *lat*, *lon*, *tz_offset*)

forecasting_schedules(*datafr*, *time*)

Returns windowlength values of given time as forecating :param datafr: schedule dataframe used to forecast :param time: current time at which DA optimization occurs

get_internal_gain_forecast(*skew_scalar*, *time*)

Forecast the electric zip_load and internal gain of all zip loads of a house by reading schedule files and applying skew. Forecast is for 48 hours ahead from start time :param skew_scalar: dictionary containing 'zip_skew', 'zip_scalar' and 'zip_heatgain_fraction' for each zip load :type skew_scalar: dict :param 'zip_skew' is a scalar and same for all type of zip loads for the given house. 'zip_scalar' and 'zip_heatgain_fraction': :param are dictionary containing different values for each type of zip load: :param time: Datetime format: forecast start time :type time: datetime

Returns
48 values of total zip loads and total internal gain due to zip loads

Return type
list

get_solar_gain_forecast(*climate_conf*, *current_time*)

get_substation_unresponsive_industrial_load_forecast(*peak_load=3500.0*)

Get substation unresponsive industrial load forecast

Args:
peak_load (float): peak load in kWh

Returns
forecast of next 48-hours unresponsive load

Return type
base_run_load (float x 48)

get_substation_unresponsive_load_forecast(*peak_load=7500.0*)

Get substation unresponsive load forecast

TODO: Update to model that make use of the base case run files TODO: Get weather forecast from weather agent

Parameters
peak_load (*float*) – peak load in kWh

Returns
forecast of next 48-hours unresponsive load

Return type
base_run_load (float x 48)

get_waterdraw_forecast(*skew_scalar*, *time*)

static initialize_schedule_dataframe(*start_time*, *end_time*)

Initialize the data frame for one year

Parameters

- **start_time** (*datetime, str*) – time in str format - DD/MM/YYY HH:MT:SS
- **end_time** (*datetime, str*) – time in str format - DD/MM/YYY HH:MT:SS

make_dataframe_schedule(*filename*, *schedule_name*)

Reads .glm files with multiple schedule names and makes dataframe for a year for given schedule name

Parameters

- **filename** (*str*) – name of glm file to be loaded
- **schedule_name** (*str*) – name of the schedule to be laoded

set_retail_price_forecast(*DA_SW_prices*)

Set substation price forecast

Nonsummable diminishing.

Parameters
DA_SW_prices (*float x 48*) – cleared price in $/kWh from the last shifting window run

Returns
forecasted prices in $/kWh for the next 48-hours

Return type
forecasted_price (float x 48)

set_sch_year(*year*)

set_solar_diffuse_forecast(*fncs_str*)

Set the 48 hour solar diffuse forecast :param solar_diffuse_forecast: :type solar_diffuse_forecast: [float x 48]

set_solar_direct_forecast(*fncs_str*)

Set the 48 hour solar direct forecast :param solar_direct_forecast: :type solar_direct_forecast: [float x 48]

tesp_support.consensus.generator module

`tesp_support.consensus.generator.`**`Consenus_dist_DA`**(*dso_market_obj*, *DA_horizon*, *fed*, *hour_of_day*, *time_granted*, *time_market_DA_complete*)

`tesp_support.consensus.generator.`**`Consenus_dist_RT`**(*dso_market_obj*, *fed*, *hour_of_day*, *time_granted*, *time_market_RT_complete*)

`tesp_support.consensus.generator.`**`construct_Laplacian`**(*N_agents*)

tesp_support.consensus.glm_dictionary module

Functions to create metadata from a GridLAB-D input (GLM) file

Metadata is written to a JSON file, for convenient loading into a Python dictionary. It can be used for agent configuration, e.g., to initialize a forecasting model based on some nominal data. It's also used with metrics output in post-processing.

Public Functions:

glm_dict
Writes the JSON metadata file.

`tesp_support.consensus.glm_dictionary.`**`append_include_file`**(*lines*, *fname*)

`tesp_support.consensus.glm_dictionary.`**`ercotMeterName`**(*objname*)

Enforces the meter naming convention for ERCOT

Replaces anything after the last _ with *mtr*.

Parameters
objname (*str*) – the GridLAB-D name of a house or inverter

Returns
The GridLAB-D name of upstream meter

Return type
str

`tesp_support.consensus.glm_dictionary.`**`glm_dict_with_microgrids`**(*name_root*, *config=None*, *ercot=False*)

Writes the JSON metadata file from a GLM file

This function reads *name_root.glm* and writes *[name_root]_glm_dict.json* The GLM file should have some meters and triplex_meters with the bill_mode attribute defined, which identifies them as billing meters that parent houses and inverters. If this is not the case, ERCOT naming rules can be applied to identify billing meters.

Parameters

- **name_root** (*str*) – path and file name of the GLM file, without the extension
- **config** (*dict*) –
- **ercot** (*bool*) – request ERCOT billing meter naming. Defaults to false. — THIS NEEDS TO LEAVE THIS PLACE
- **te30** (*bool*) – request hierarchical meter handling in the 30-house test harness. Defaults to false. — THIS NEEDS TO LEAVE THIS PLACE

`tesp_support.consensus.glm_dictionary.`**`ti_enumeration_string`**(*tok*)

if thermal_integrity_level is an integer, convert to a string for the metadata

tesp_support.consensus.microgrid module

`tesp_support.consensus.microgrid.`**`Consenus_dist_DA`**(*dso_market_obj*, *DA_horizon*, *fed*, *time_granted*, *time_market_DA_complete*)

`tesp_support.consensus.microgrid.`**`Consenus_dist_RT`**(*dso_market_obj*, *fed*, *time_granted*, *time_market_RT_complete*)

`tesp_support.consensus.microgrid.`**`construct_Laplacian`**(*N_agents*)

tesp_support.consensus.microgrid_agent module

Manages the Transactive Control scheme for DSO+T implementation version 1

Public Functions:

substation_loop
initializes and runs the agents

`tesp_support.consensus.microgrid_agent.`**`destroy_federate`**(*fed*)

`tesp_support.consensus.microgrid_agent.`**`inner_substation_loop`**(*configfile*, *metrics_root*, *with_market*)

Helper function that initializes and runs the DSOT agents

TODO: This needs to be updated Reads configfile. Writes *auction_metrics_root_metrics.json* and *controller_metrics_root_metrics.json* upon completion.

Parameters

- **`configfile`** (*`str`*) – fully qualified path to the JSON agent configuration file
- **`metrics_root`** (*`str`*) – base name of the case for metrics output
- **`with_market`** (*`bool`*) – flag that determines if we run with markets

`tesp_support.consensus.microgrid_agent.`**`register_federate`**(*json_filename*)

`tesp_support.consensus.microgrid_agent.`**`substation_loop`**(*configfile*, *metrics_root*, *with_market=True*)

Wrapper for *inner_substation_loop*

When *inner_substation_loop* finishes, timing and memory metrics will be printed for non-Windows platforms.

`tesp_support.consensus.microgrid_agent.`**`worker`**(*arg*)

tesp_support.consensus.residential_feeder_glm module

tesp_support.consensus.retail_market module

Class that manages the operation of retail market at substation-level

Functionalities include: collecting curve bids from DER and DSO agents; generating aggregated buyer and seller curves; market clearing for both RT and DA retail markets; deciding cleared quantity for individual DERs.

The function call order for this agent is:

initialize(retail_dict)

Repeats at every hour:
clean_bids_DA() curve_aggregator_DA(identity, bid, id) … curve_aggregator_DA(identity, bid, id) clear_market_DA(transformer_degradation, Q_max)

Repeats at every 5 min:
clean_bids_RT() curve_aggregator_RT(identity, bid, id) … curve_aggregator_RT(identity, bid, id) clear_market_RT(transformer_degradation, Q_max)

class `tesp_support.consensus.retail_market.`**`RetailMarket`**(*retail_dict*, *key*)

Bases: `object`

This agent manages the retail market operating

Parameters

- **`retail_dict`** –
- **`key`** –

`name`

name of the retail market agent

Type
str

`price_cap`

the maximum price that is allowed in the market, in $/kWh

Type
float

`num_samples`

the number of sampling points, describes how precisely the curve is sampled

Type
int

`windowLength`

length of the planning horizon for the DA market, in hours

Type
int

`Q_max`

capacity of the substation, in kWh

Type
float

`maxPuLoading`

rate of the maxPuLoading for transformer

Type
float

`curve_buyer_RT`

aggregated buyer curve, updated after receiving each RT buyer bid

Type
Curve

curve_seller_RT

aggregated seller curve, updated after receiving each RT seller bid

Type
Curve

curve_buyer_DA

48 aggregated buyer curves, updated after receiving each DA buyer bid

Type
dict of curves

curve_seller_DA

48 aggregated seller curves, updated after receiving each DA seller bid

Type
dict of curves

clear_type_RT

0=uncongested, 1=congested, 2=inefficient, 3=failure

Type
int

clear_type_DA

list of clear type at each hour

Type
list

cleared_price_RT

cleared price for the substation for the next 5-min

Type
float

cleared_price_DA

list of cleared price at each hour

Type
list

cleared_quantity_RT

cleared quantity for the substation for the next 5-min

Type
float

cleared_quantity_DA

list of cleared quantity at each hour

Type
list

site_responsive_DA

Site_Day_Ahead quantity which is responsive

Type
list

site_unresponsive_DA

Site Day Ahead quantity which is unresponsive

Type
list

AMES_RT

Smooth Quadratics

Type
list X 5

AMES_DA

Smooth Quadratics

Type
(list X 5) X windowLength

TOC_dict

parameters related to transformer lifetime cost calculation, including OperatingPeriod (int): operating period, in minute timeStep (int): timestep, in minute Tamb (float): ambient temperature, in deg C delta_T_TO_init (int): initial delta temperature of top oil, in deg C delta_T_W_init (int): initial delta temperature of winding, in deg C BP (float): initial cost of transformer, in $ toc_A (float): cost per watt for no-load losses, in $/W toc_B (float): cost per watt for load losses, in $/W Base_Year (float): expected lifespan of transformer, in year P_Rated (float): capacity, in W NLL_rate (float): no load loss rate, in % LL_rate (float): load loss rate, in % Sec_V (float): secondary voltage level, in volt TOU_TOR (float): oil time constant, in minute TOU_GR (float): winding time constant, in minute Oil_n (float): Oil exponent n Wind_m (float): Winding exponent m delta_T_TOR (float): top oil temperature rise, in deg C delta_T_ave_wind_R (float): average winding temperature rise over ambient temperature, in deg C

Type
dict

clean_bids_DA()

Initialize the day-ahead market

clean_bids_RT()

Initialize the real-time market

clear_market(*curve_buyer*, *curve_seller*, *transformer_degradation*, *Q_max*)

Shared function called by both clear_market_RT and clear_market_DA functions to find the intersection between supply curve and demand curve

Parameters

- **curve_buyer** (Curve) – aggregated buyer curve
- **curve_seller** (Curve) – aggregated seller curve
- **transformer_degradation** (*bool*) – equals to 1 if transformer_degradation is considered in the supply curve
- **Q_max** (*float*) – substation capacity, in kWh

Outputs:
clear_type (int) cleared_price (float) cleared_quantity (float) congestion_surcharge (float)

clear_market_DA(*transformer_degradation*, *Q_max*)

Function used for clearing the DA market

Three steps of work are fullfilled in a loop in this function: First the buyer curve at each hour is fitted polynomial; Second clear_market function is called for calculating the cleared price and cleared quantity for the whole market at each hour; Third distribute_cleared_quantity function is called for finding the cleared price and cleared quantity for the individual DERs at each hour.

buyer_info_DA and seller_info_DA, clear_type_DA, cleared_price_DA and cleared_quantity_DA are updated with cleared results

clear_market_RT(*transformer_degradation*, *Q_max*)

Function used for clearing the RT market

Three steps of work are fullfilled in this function: First the buyer curve is fitted polynomial; Second clear_market function is called for calculating the cleared price and cleared quantity for the whole market; Third distribute_cleared_quantity function is called for finding the cleared price and cleared quantity for the individual DERs.

buyer_info_RT and seller_info_RT, clear_type_RT, cleared_price_RT and cleared_quantity_RT are updated with cleared results

convert_2_AMES_quadratic_BID(*curve*, *Q_cleared*, *price_forecast*)

Convert aggregated DSO type bid to AMES quadratic curve

Parameters

- **curve** (Curve) – substation demand curve to be preprocessed
- **price_forecast** – locally forecast price at the substation level
- **Q_cleared** – locally cleared quantity at the substation

Returns

f(Q)=resp_c2*Q^2+C1*Q+C0
unresp_mw (float): minimum demand MW resp_max_mw (float): maximum demand MW resp_c2 (float): quadratic coefficient resp_c1 (float): linear coefficient resp_c0 (float): constant coefficient resp_deg (int): equal to "2" to represent the current order in the list

Return type
quadratic_bid (list)

curve_aggregator_AMES_DA(*demand_curve_DA*, *Q_max*, *Q_cleared*, *price_forecast*)

Function used to aggregate the substation-level DA demand curves into a DSO-level DA demand curve

Parameters

- **demand_curve_DA** (*dict*) – a collection of demand curves to be aggregated for day-ahead
- **Q_max** (*float*) – maximum capacity of the substation, in kW
- **Q_cleared** (*float*) – locally cleared quantity of the substation in kW
- **price_forecast** (*float*) – locally forecast price at the substation level

curve_aggregator_AMES_RT(*demand_curve_RT*, *Q_max*, *Q_cleared*, *price_forecast*)

Function used to aggregate the substation-level RT demand curves into a DSO-level RT demand curve

Parameters

- **demand_curve_RT** (Curve) – demand curve to be aggregated for real-time
- **Q_max** (*float*) – maximum capacity of the substation, in kW

- **Q_cleared** (*float*) – locally cleared quantity of the substation in kW
- **price_forecast** (*float*) – locally forecast price at the substation level

curve_aggregator_DA(*identity*, *bid_DA*, *name*)

Function used to collect the DA bid and update the accumulated buyer or seller curve

Parameters

- **identity** (*str*) – identifies whether the bid is collected from a "Buyer" or "Seller"
- **bid_DA** (*list*) – a nested list with dimension (self.windowLength, m, 2), with m equals 2 to 4
- **name** (*str*) – name of the buyer or seller

curve_aggregator_RT(*identity*, *bid_RT*, *name*)

Function used to collect the RT bid and update the accumulated buyer or seller curve

Parameters

- **identity** (*str*) – identifies whether the bid is collected from a "Buyer" or "Seller"
- **bid_RT** (*list*) – a nested list with dimension (m, 2), with m equals 2 to 4
- **name** (*str*) – name of the buyer or seller

curve_preprocess(*substation_demand_curve*, *Q_max*)

An internal shared function called by curve_aggregator_DSO_RT and curve_aggregator_DSO_DA functions to truncate

the substation demand curve before aggregation as well as convert the retail prices into wholesale prices

Parameters

- **substation_demand_curve** (Curve) – substation demand curve to be preprocessed
- **Q_max** (*float*) – maximum capacity of the substation, in kW

Returns

preprocessed demand curve

Return type

preprocessed_curve (*curve*)

formulate_bid_industrial_da(*industrial_load_forecast*, *price_forecast*)

This file is to create a bid curve for industrial customer for the given demand-price elasticity

elasticity (a) = (del-Q/Q)/(del-P/P) –> del-Q/del-p = a * Q / P The slope of the bid curve for the industrial load will be Slope (S) = (del-Q/del-P) The Q-axis intercept for the bid curve will be (C_Q) = Q + S * del-p = Q - a * Q The P-axis intercept for the bid curve will be (C_P) = P + S * 1 / del-Q = P - P / a

Args:

price_forecast: forecasted price in $/kWh at the instance (if we are going to bid directly in ISO, then this is LMP forecast, or else, it is retail price forecast industrial_load_forecast: forecast quantity of industrial load in kW at that instant

Returns:

bid_da (float) (1 x windowLength): Bid quantity from optimization for all hours of the window specified by windowLength

formulate_bid_industrial_rt(*industrial_load*, *price_cleared*)

This file is to create real time bid curve for industrial customer for the given demand-price elasticity

elasticity (a) = (del-Q/Q)/(del-P/P) –> del-Q/del-p = a * Q / P The slope of the bid curve for the industrial load will be Slope (S) = (del-Q/del-P) The Q-axis intercept for the bid curve will be (C_Q) = Q + S * del-p = Q - a * Q The P-axis intercept for the bid curve will be (C_P) = P + S * 1 / del-Q = P - P / a

Args:
price_cleared (float): Price cleared in $/kWh at the instance (if we are going to bid directly in ISO, then this is LMP forecast, or else, it is retail price forecast industrial_load (float): The most updated (closer to real-time) industrial load in kW at that instant

Returns:
bid_rt (1,2)X4): 4 point industrial load bid

process_site_da_quantities(*forecast_load*, *name*, *status*)

Function stores the day-ahead quantities, primarily for HVAC at the moment, it utilizes

arguments in: unresponsive loads (1x 48) dataframe responsive loads (1x48) dataframe name: of the house (str) returns self.site_quantity_DA (dict, 'name', ' status (participating or not participating', 'Quantity (1 x 48))

retail_rate_inverse(*Pr*)

Function used to convert the retail prices into wholesale prices

Parameters
Pr (*float*) – retail price, in $/kWh

Returns
wholesale price, in $/kWh

Return type
Pw (float)

test_function()

Test function with the only purpose of returning the name of the object

tesp_support.consensus.retail_market.**test**()

Testing AMES

tesp_support.consensus.substation module

tesp_support.consensus.substation.**Consenus_dist_DA**(*dso_market_obj*, *DA_horizon*, *fed*, *hour_of_day*, *time_granted*, *time_market_DA_complete*)

tesp_support.consensus.substation.**Consenus_dist_RT**(*dso_market_obj*, *fed*, *hour_of_day*, *time_granted*, *time_market_RT_complete*)

tesp_support.consensus.substation.**construct_Laplacian**(*N_agents*)

tesp_support.consensus.weather_agent module

Weather Agent

This weather agent needs an WEATHER_CONFIG environment variable to be set, which is a json file.

`tesp_support.consensus.weather_agent.convertTimeToSeconds(time)`

Convert time string with unit to integer in seconds

It only parse unit in day, hour, minute and second. It will not recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
(int) represent the input time in second

`tesp_support.consensus.weather_agent.deltaTimeToResmapleFreq(time)`

Convert time unit to a resampling frequency that can be recognized by pandas.DataFrame.resample()

It only parse unit in day, hour, minute and second. It won't recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
(str) time with resample frequency

`tesp_support.consensus.weather_agent.destroy_federate(fed)`

`tesp_support.consensus.weather_agent.findDeltaTimeMultiplier(time)`

Find the multiplier to convert delta_time to seconds

It only parse unit in day, hour, minute and second. It won't recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
(int) the multiplier to convert delta_time to seconds

`tesp_support.consensus.weather_agent.register_federate(json_filename)`

`tesp_support.consensus.weather_agent.startWeatherAgent(file)`

The weather agent publishes weather data as configured by the json file

Parameters
file (*str*) – the weather data file

Returns
(none)

`tesp_support.consensus.weather_agent.usage()`

class `tesp_support.consensus.weather_agent.weather_forecast(variable, period, W_dict)`

Bases: `object`

This object includes the error to a weather variable

Parameters

- **variable** (*str*) – Type of weather variable being forecasted
- **period** (*int*) – period of the sinusoidal bias
- **W_dict** (*dict*) – dictionary for specifying the generation of the error envelope

weather_variable

Type of weather variable being forecasted

Type
str

 # Type of error insertion

distribution

type of distribution –> 0 uniform;1 triangular;2 truncated normal the standard deviation is computed for 95% of values to be within bounds in a conventional normal distribution

Type
int

P_e_bias

pu maximum bias at first hour –> [0 to 1]

Type
float

P_e_envelope

pu maximum error from mean values –> [0 to 1]

Type
float

Lower_e_bound

pu of the maximum error at the first hour –> [0 to 1]

Type
float

 # Bias variable

biasM

sinusoidal bias for altering the error envelope

Type
float) (1 X period

Period_bias

period of the sinusoidal bias

Type
int

get_truncated_normal(*EL*, *EH*)

Truncated normal distribution

make_forecast(*weather*, *t=0*)

Include error to a known weather variable

Parameters

- **weather** (*float) (1 x desired number of hours ahead*) – known weather variable

- **t** (*int*) – time in hours

Returns
weather variable with included error ENV_U (float) (1 x desired number of hours ahead): envelope with bias upper bound ENV_l (float) (1 x desired number of hours ahead): envelope with bias lower bound

Return type
weather_f (float) (1 x desired number of hours ahead)

tesp_support.dsot package

Transactive Energy Simulation Platform (TESP) Contains the python files for the DSOT analysis

Submodules

tesp_support.dsot.Wh_Energy_Purchases module

Utilities to open and read load

Todo: files should be passed in for 'load*' functions

Public Functions:

load_hourly_data
Utility to open csv file for hourly load data

load_realtime_data
Utility to open csv file 5 min load data

load_price_data
Utility to open csv file LMP data

Wh_Energy_Purchases
Computes the total costs, total energy purchases and average price annually

`tesp_support.dsot.Wh_Energy_Purchases.`**`Wh_Energy_Purchases`**(*dir_path*, *dso_num*, *simdata=False*, *h1=5*, *h2=16*, *h3=20*, *place='Houston'*)

Computes the total costs, total energy purchases and average price annually for bilateral, day-ahead and real-time markets from hourly and 5 min ERCOT energy and price data.

- Loads hourly ERCOT energy and price data.
- Computes minimum wholesale price and quantity for day, evening, and weekend to represent the fixed prices and energy quantities for the bilateral market.
- Total annual average price, total cost and total energy purchases from the bilateral market are then computed.

Parameters

- **dir_path** (*str*) – path of the ERCOT energ and price data
- **dso_num** (*int*) – dso number (1-8)
- **simdata** (*bool*) – If True will seek simulation data. If false will use 2016 ERCOT data.
- **h1** (*int*) – hours specified to define day
- **h2** (*int*) – hours specified to define evening

- **h3** (*int*) – hours specified to define night respectively (e.g. weekday hours h1 to h2, evening hours from h2+1 to h3, and night hours are from h3+1 to h1 the next day)
- **place** (*str*) – location of DSO

Returns
real-time, day-ahead, and bilateral market purchases (annual cost, energy and average price)

Return type
dict

`tesp_support.dsot.Wh_Energy_Purchases.`**`load_hourly_data`**(*dir_path*, *dso_num*, *simdata*)

Utility to open hourly ERCOT csv file.

The entire date range for the data (e.g. 1 year) is considered

Parameters

- **dir_path** (*str*) – directory path of data
- **dso_num** (*int*) – dso number
- **simdata** (*bool*) – If True will seek simulation data. If false will use 2016 ERCOT data.

Returns
dataframe of ERCOT data for dso specified

Return type
dataframe

`tesp_support.dsot.Wh_Energy_Purchases.`**`load_price_data`**(*dir_path*, *market_type*, *dso_num*, *simdata*, *place*)

Utility to open 5 min ERCOT csv file.

The entire date range for the data (e.g. 1 year) is considered

Parameters

- **dir_path** (*str*) – directory path of data
- **market_type** (*str*) – string telling the market type ('DA','RT')
- **dso_num** (*int*) – dso number
- **simdata** (*bool*) – If True will seek simulation data. If false will use 2016 ERCOT data.
- **place** (*str*) – location of DSO

Return type
prices_data

`tesp_support.dsot.Wh_Energy_Purchases.`**`load_realtime_data`**(*dir_path*, *dso_num*, *simdata*)

Utility to open 5 min ERCOT csv file.

The entire date range for the data (e.g. 1 year) is considered

Parameters

- **dir_path** (*str*) – directory path of data
- **dso_num** (*int*) – dso number
- **simdata** (*bool*) – If True will seek simulation data. If false will use 2016 ERCOT data.

Returns
15 min dataframe of ERCOT data for specified dso

Return type
dataframe

tesp_support.dsot.balance_sheet_functions module

`tesp_support.dsot.balance_sheet_functions.iso_balance_sheet_annual(meta_path, write_path, write_to_txt=False, write_to_JSON=False)`

This function calculates …

Parameters

- **meta_path** –
- **write_path** –
- **write_to_txt** –
- **write_to_JSON** –

Return type
dict

`tesp_support.dsot.balance_sheet_functions.test_iso()`

`tesp_support.dsot.balance_sheet_functions.test_too()`

`tesp_support.dsot.balance_sheet_functions.too_balance_sheet_annual(meta_path, write_path, write_to_txt=False, write_to_JSON=False)`

This function calculates …

Parameters

- **meta_path** –
- **write_path** –
- **write_to_txt** –
- **write_to_JSON** –

Return type
dict

tesp_support.dsot.battery_agent module

Class that controls the Battery DER

Implements the optimum schedule of charging and discharging DA; generate the bids for DA and RT; monitor and supervisory control of GridLAB-D environment element.

The function call order for this agent is:

- initialize()

Repeats at every hour

- formulate_bid_da() {return BID}
- set_price_forecast(forecasted_price)

Repeats at every 5 mins

- set_battery_SOC(msg_str) {updates C_init}
- formulate_bid_rt() {return BID}
- inform_bid(price) {update RTprice}
- bid_accepted() {update inv_P_setpoint and GridLAB-D P_out if needed}

class tesp_support.dsot.battery_agent.**BatteryDSOT**(*battery_dict*, *inv_properties*, *key*, *model_diag_level*, *sim_time*, *solver*)

Bases: `object`

This agent manages the battery/inverter

Parameters

- **battery_dict** (*dict*) –
- **inv_properties** (*dict*) –
- **key** (*str*) –
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable
- **solver** (*str*) –

Initialize from Args

name

name of this agent

Type
str

Rc

rated charging power in kW for the battery

Type
float

Rd

rated discharging power in kW for the battery

Type
float

Lin

battery charging loss in %

Type
float

Lout

battery discharging loss in %

Type
float

Cmin

minimum allowable stored energy in kWh (state of charge lower limit)

Type
float

Cmax

maximum allowable stored energy in kWh (state of charge upper limit)

Type
float

Cinit

initial stored energy in the battery in kWh

Type
float

batteryCapacity

battery capacity in kWh

Type
float

batteryLifeDegFactor

constant to model battery degradation

Type
float

windowLength

length of day ahead optimization period in hours (e.g. 48-hours)

Type
int

dayAheadCapacity

% of battery capacity reserved for day ahead bidding

Type
float

No initialization required

bidSpread

this can be used to spread out bids in multiple hours. When set to 1 hour (recommended), it's effect is none

Type
int

P

location of P in bids

Type
int

Q

location of Q in bids

Type
int

f_DA

forecasted prices in $/kWh for all the hours in the duration of windowLength

Type
List[float]) (1 X windowLength

ProfitMargin_slope

specified in % and used to modify slope of bid curve. Set to 0 to disable

Type
float

ProfitMargin_intercept

specified in % to generate a small dead band (i.e., change in price does not affect quantity). Set to 0 to disable

Type
float

pm_hi

Highest possible profit margin in %

Type
float

pm_lo

Lowest possible profit margin in %

Type
float

RT_state_maintain

true if battery must maintain charging or discharging state for 1 hour

Type
bool

RT_state_maintain_flag

(0) not define at current hour (-1) charging (+1) discharging

Type
int

RT_flag

if True, has to update GridLAB-D

Type
bool

inv_P_setpoint

next GridLAB-D inverter power output

Type
float

optimized_Quantity

Optimized quantity

Type
List[float]) (1 X Window Length

#not used if not biding DA

prev_clr_Quantity

cleared quantities (kWh) from previous market iteration for all hours

Type
List[float]) (1 X Window Length

prev_clr_Price

cleared prices ($/kWh) from previous market iteration

Type
List[float]) (1 X windowLength

BindingObjFunc

if True, then optimization considers cleared price, quantities from previous iteration in the objective function

Type
bool

DA_cleared_price(*price*)

Set the DA_cleared_price attribute

Parameters
price (*float*) – cleared price in $/kWh

DA_optimal_quantities()

Generates Day Ahead optimized quantities for Battery

Returns
Optimal quantity from optimization for all hours of the window specified by windowLength

Return type
Quantity (float) (1 x windowLength)

RT_fix_four_points_range(*BID*, *Ql*, *Qu*)

Verify feasible range of RT bid

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **Ql** –
- **Qu** –

Returns
4 point bid only the feasible range

Return type
BIDr (float) ((1,2)X4)

RT_gridlabd_set_P(*model_diag_level*, *sim_time*)

Update variables for battery output "inverter"

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

inv_P_setpoint is a float in W

bid_accepted(*current_time*)

Update the P and Q settings if the last bid was accepted

Returns
True if the inverter settings changed, False if not.

Return type
bool

con_rule_eq1(*m*, *i*)

con_rule_eq2(*m*, *i*)

con_rule_eq3(*m*, *i*)

con_rule_ine1(*m*, *i*)

con_rule_ine2(*m*, *i*)

formulate_bid_da()

Formulate 4 points of P and Q bids for the DA market

Function calls "DA_optimal_quantities" to obtain the optimal quantities for the DA market. With the quantities, the 4 point bids are formulated.

Before returning the BID the function resets "RT_state_maintain_flag" wich if RT_state_maintain is TRUE the battery will be forced to keep its state (i.e., charging or discharging).

Returns
store last DA market bids

Return type
BID (float) (((1,2)X4) X windowLength)

formulate_bid_rt()

Formulates RT bid

Uses the last 4 point bid from DA market and consider current state of charge of the battery. Will change points to change points for feasible range of Qmin Qmax points if necessary. Furthermore, allows a maximum deviation of +/-100% from the DA plan.

Returns
bid in Real Time market

Return type
realTimeBid (float) ((1,2) x 4)

from_P_to_Q_battery(*BID*, *PRICE*)

Convert the 4 point bids to a quantity with the known price

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **PRICE** (*float*) – cleared price in $/kWh

Returns
active power (-) charging (+) discharging

Return type
_quantity (float)

inform_bid(*price*)

Set the cleared_price attribute

Parameters

price (*float*) – cleared price in $/kWh

obj_rule(*m*)

set_battery_SOC(*msg_str*, *model_diag_level*, *sim_time*)

Set the battery state of charge

Updates the self.Cinit of the battery

Parameters

- **msg_str** (*str*) – message with battery SOC in pu
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

set_price_forecast(*forecasted_price*)

Set the f_DA attribute

Parameters

forecasted_price (*float x 48*) – cleared price in $/kWh

test_function()

Test function with the only purpose of returning the name of the object

tesp_support.dsot.battery_agent.**test**()

Makes a single battery agent and run DA

tesp_support.dsot.case_comparison_plots module

tesp_support.dsot.case_comparison_plots.**plot_lmp_stats**(*cases*, *data_paths*, *output_path*, *dso_num*, *variable*)

Will plot LMPS by month, duration, and versus netloads loads (for select month), and save to file. :param cases: names of the cases :type cases: List[str] :param data_paths: location of the data files to be used. :type data_paths: str :param output_path: path of the location where output (plots, csv) should be saved :type output_path: str :param dso_num: bus number for LMP data to be plotted :type dso_num: str :param variable: :type variable: str

Returns

saves dso lmps plots to file

tesp_support.dsot.case_merge module

Combines GridLAB-D and agent files to run a multi-feeder TESP simulation

Public Functions:

merge_glm

combines GridLAB-D input files

merge_glm_dict

combines GridLAB-D metadata files

merge_agent_dict

combines the substation agent configuration files

merge_substation_yaml
combines the substation agent FNCS publish/subscribe files

merge_fncs_config
combines GridLAB-D FNCS publish/subscribe files

`tesp_support.dsot.case_merge.`**`merge_agent_dict`**(*target*, *sources*)

Combines the substation agent configuration files into "target". The source files must already exist.

Parameters

- **target** (*str*) – the path to the target JSON file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.dsot.case_merge.`**`merge_fncs_config`**(*target*, *sources*)

Combines GridLAB-D input files into "target". The source feeders must already exist.

Parameters

- **target** (*str*) – the path to the target TXT file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.dsot.case_merge.`**`merge_glm`**(*target*, *sources*, *xfmva*)

Combines GridLAB-D input files into "target". The source files must already exist.

Parameters

- **target** (*str*) – the path to the target GLM file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge
- **xfmva** (*int*) –

`tesp_support.dsot.case_merge.`**`merge_glm_dict`**(*target*, *sources*, *xfmva*)

Combines GridLAB-D metadata files into "target". The source files must already exist.

The output JSON won't have a top-level base_feeder attribute. Instead, the base_feeder from each source file will become a feeder key in the output JSON feeders dictionary, and then every child object on that feeder will have its feeder_id, originally network_node, changed to match the base_feeder.

Parameters

- **target** (*str*) – the path to the target JSON file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge
- **xfmva** (*int*) –

`tesp_support.dsot.case_merge.`**`merge_substation_yaml`**(*target*, *sources*)

Combines GridLAB-D input files into "target". The source files must already exist.

Parameters

- **target** (*str*) – the path to the target YAML file, including the name of the file
- **sources** (*list*) – list of feeder names in the target directory to merge

tesp_support.dsot.customer_CFS module

@author: yint392

tesp_support.dsot.customer_CFS.**customer_CFS**(*GLD_metadata*, *metadata_path*, *customer*, *customer_bill*)

tesp_support.dsot.customer_CFS.**get_customer_df**(*dso_range*, *case_path*, *metadata_path*)

tesp_support.dsot.dso_CFS module

@author: yint392

tesp_support.dsot.dso_CFS.**dso_CFS**(*case_config*, *DSOmetadata*, *dso_num*, *DSO_peak_demand*, *DSO_base_case_peak_demand*, *DSO_Cash_Flows*, *DSO_Revenues_and_Energy_Sales*, *Market_Purchases*, *Market_Purchases_base_case*)

tesp_support.dsot.dso_CFS.**get_DSO_df**(*dso_range*, *case_config*, *DSOmetadata*, *case_path*, *base_case_path*)

tesp_support.dsot.dso_helper_functions module

@author: yint392

tesp_support.dsot.dso_helper_functions.**TEAM**(*FteLev1=100.0*, *SalaryEsc1=1.3*)

tesp_support.dsot.dso_helper_functions.**get_mean_for_diff_groups**(*df*, *main_variables*, *variables_combs*, *cfs_start_position=24*)

tesp_support.dsot.dso_helper_functions.**labor**(*group*, *metadata_general*, *metadata_dso*, *utility_type*, *NoSubstations*)

tesp_support.dsot.dso_helper_functions.**labor_increase**(*group*, *metadata_general*, *metadata_dso*, *utility_type*, *NoSubstations*, *TransactiveCaseFlag*)

tesp_support.dsot.dso_helper_functions.**labor_network_admin**(*group*, *hourly_rate*, *metadata_general*, *metadata_dso*, *utility_type*, *NoSubstations*)

tesp_support.dsot.dso_helper_functions.**labor_network_admin_increase**(*group*, *hourly_rate*, *metadata_general*, *metadata_dso*, *utility_type*, *NoSubstations*, *TransactiveCaseFlag*)

tesp_support.dsot.dso_helper_functions.**labor_network_admin_transactive**(*group*, *hourly_rate*, *metadata_general*, *metadata_dso*, *utility_type*, *NoSubstations*, *TransactiveCaseFlag*)

`tesp_support.dsot.dso_helper_functions.labor_transactive(group, metadata_general, metadata_dso, utility_type, NoSubstations, TransactiveCaseFlag)`

`tesp_support.dsot.dso_helper_functions.returnDictSum(temp_dict)`

tesp_support.dsot.dso_map module

`tesp_support.dsot.dso_map.prepare_metadata(node, end_row, feeder_mode, high_renewables_case, DSO_Load_Threshold)`

tesp_support.dsot.dso_market module

Class that manages the operation of DSO agent

Functionalities include: Aggregate demand bids from different substations; Wholesale no da trial clearing; Conversion between wholesale price and retail price; Generate substation supply curves with and without consideration of the transformer degradation.

class `tesp_support.dsot.dso_market.DSOMarket(dso_dict, key)`

Bases: `object`

This agent manages the DSO operating

Parameters

- **dso_dict** –
- **key** –

name

name of the DSO agent

Type
str

price_cap

the maximum price that is allowed in the market, in $/kWh

Type
float

num_samples

the number of sampling points, describes how precisely the curve is sampled

Type
int

windowLength

length of the planning horizon for the DA market, in hours

Type
int

DSO_Q_max

maximum limit of the DSO load capacity, in kWh

Type
float

transformer_degradation
flag variable, equals to 1 when transformer degradation effect is taken into account

Type
bool

curve_a
array of second order coefficients for the wholesale node curve, indexed by day_of_sim and hour_of_day

Type
array

curve_b
array of first order coefficients of the wholesale node curve, indexed by day_of_sim and hour_of_day

Type
array

curve_c
array of intercepts of the wholesale node curve, indexed by day_of_sim and hour_of_day

Type
array

Pwclear_RT
cleared wholesale price by real-time wholesale node trial clearing, in $/kWh

Type
float

Pwclear_DA
list of cleared wholesale price by day-ahead wholesale node trial clearing, in $/kWh, indexed by hour

Type
list

trial_cleared_quantity_RT
trial cleared quantity by real-time wholesale node trial clearing, in kWh

Type
float

trial_cleared_quantity_DA
trial cleared quantity by day-ahead wholesale node trial clearing, in kWh

Type
list

curve_DSO_RT
aggregated demand curve at DSO level from real-time retail market

Type
Curve

curve_DSO_DA
dictionary of aggregated demand curves at DSO level from day-ahead retail market, indexed by hour

Type
dict

curve_ws_node

dictionary of wholesale node curves, indexed by day_of_sim and hour_of_day

Type
dict

trial_clear_type_RT

trial cleared type of real-time wholesale node trial clearing

Type
int

trial_clear_type_DA

trial cleared type of day-ahead wholesale node trial clearing, indexed by hour

Type
list

hour_of_day

current hour of the day

Type
int

day_of_week

current day of the week

Type
int

customer_count_mix_residential

Residential percentage of the total customer count mix

number_of_gld_homes

Total number of GLD homes for the DSO

clean_bids_DA()

Initialize the day-ahead wholesale node trial clearing

clean_bids_RT()

Initialize the real-time wholesale node trial clearing

curve_aggregator_DSO_DA(*demand_curve_DA*, *Q_max*)

Function used to aggregate the substation-level DA demand curves into a DSO-level DA demand curve

Parameters

- **demand_curve_DA** (*dict*) – a collection of demand curves to be aggregated for day-ahead
- **Q_max** (*float*) – maximum capacity of the substation, in kW

curve_aggregator_DSO_RT(*demand_curve_RT*, *Q_max*)

Function used to aggregate the substation-level RT demand curves into a DSO-level RT demand curve

Parameters

- **demand_curve_RT** (Curve) – demand curve to be aggregated for real-time
- **Q_max** (*float*) – maximum capacity of the substation, in kW

curve_preprocess(*substation_demand_curve*, *Q_max*)

An internal shared function called by curve_aggregator_DSO_RT and curve_aggregator_DSO_DA functions to truncate the substation demand curve before aggregation as well as convert the retail prices into wholesale prices

Parameters

- **substation_demand_curve** (Curve) – substation demand curve to be preprocessed
- **Q_max** (*float*) – maximum capacity of the substation, in kW

Returns
preprocessed demand curve

Return type
preprocessed_curve (*curve*)

generate_TOC(*costInterval*, *maxPuLoad*, *num_samples*, *TOC_dict*)

Function used to calculate the total owning cost of transformer

Parameters

- **costInterval** (*int*) – interval for calculating the cost, in minutes
- **maxPuLoad** (*float*) – maximum pu loading factor
- **num_samples** (*int*) – number of sampling points
- **TOC_dict** (*dict*) – configuration parameters for transformer

Returns
price axis of unit owning cost, in $/kWh LoadsForPlot (list): quantity axis of unit owing cost, in kWh

Return type
DollarsForPlot (list)

get_prices_of_quantities(*Q*, *day*, *hour*)

Returns the prices DSO quadratic curve cost for a list of quantities

Parameters

- **Q** (*list of float*) – quantities
- **day** (*int*) – day of the week
- **hour** (*int*) – hour of the day

Returns
prices for the quantities

Return type
P (list of float)

retail_rate(*Pw*)

Function used to convert the wholesale prices into retail prices

Parameters
Pw (*float*) – wholesale price, in $/kWh

Returns
retail price, in $/kWh

Return type
Pr (float)

retail_rate_inverse(*Pr*)

Function used to convert the retail prices into wholesale prices

Parameters
Pr (*float*) – retail price, in $/kWh

Returns
wholesale price, in $/kWh

Return type
Pw (float)

set_Pwclear_DA(*hour_of_day*, *day_of_week*)

Function used to implement the DA trial wholesale node clearing and update the Pwclear_DA value

Parameters

- **hour_of_day** (*int*) – current hour of the day
- **day_of_week** (*int*) – current day of the week

set_Pwclear_RT(*hour_of_day*, *day_of_week*, *lmp=False*)

Function used to implement the RT trial wholesale node clearing and update the Pwclear_RT value

Parameters

- **hour_of_day** (*int*) – current hour of the day
- **day_of_week** (*int*) – current day of the week
- **lmp** –

set_cleared_q_da(*val*)

Set the clear quantity for day ahead

Parameters
val (*double*) – lmp for the bus/substation

set_cleared_q_rt(*val*)

Set the clear quantity for real time

Parameters
val (*double*) – lmp for the bus/substation

set_ind_load(*industrial_load*)

Set the industrial load based on provided load by a csv file after base-case, complex number

Parameters
industrial_load (*str*) – industrial load of substation

set_ind_load_da(*industrial_load_da*)

Set the ercot ind load for the next 24-hours provided load by a csv file after base-case, complex number

Parameters
industrial_load_da (*24 x 1 array of double*) – industry load values for the next day ahead are given in MW

set_lmp_da(*val*)

Set the lmp for day ahead

Parameters
val (*array of double*) – lmp for the day ahead

set_lmp_rt(*val*)

Set the lmp for real time

Parameters
val (*double*) – lmp for the bus/substation

set_ref_load(*ref_load*)

Set the reference (ercot) load based on provided load by a csv file after base-case, complex number

Parameters
ref_load (*str*) – total load of substation

set_ref_load_da(*ref_load_da*)

Set the ercot load for the next 24-hours provided load by a csv file after base-case, complex number

Parameters
ref_load_da (*24 x 1 array of double*) – ercot load values for the next day ahead

set_total_load(*total_load*)

Set the residential load based on provided load by GLD, complex number

Parameters
total_load (*str*) – total load of substation

substation_supply_curve_DA(*retail_obj*)

Function used to generate the DA supply curves for each substation

Args:

Variables:
FeederCongPrice (float): feeder congestion price, in $/kWh FeederPkDemandPrice (float): feeder peak demand price, in $/kWh FeederCongCapacity (float): feeder congestion capacity, in kWh FeederPkDemandCapacity (float): feeder peak demand, in kWh Q_max_retail (float): substation limit, in kWh Q_max_DSO (float): can change when the demand bid is higher than the original DSO limit maxPuLoading (float): maximum pu loading factor TOC_dict (dict): configuration parameters for transformer

Returns
a collection of substation supply curves for day-ahead market clearing

Return type
supply_curve_DA (list)

substation_supply_curve_RT(*retail_obj*)

Function used to generate the RT supply curve for each substation

Args:

Variables:
FeederCongPrice (float): feeder congestion price, in $/kWh FeederPkDemandPrice (float): feeder peak demand price, in $/kWh FeederCongCapacity (float): feeder congestion capacity, in kWh FeederPkDemandCapacity (float): feeder peak demand, in kWh Q_max_retail (float): substation limit, in kWh Q_max_DSO (float): can change when the demand bid is higher than the original DSO limit maxPuLoading (float): maximum pu loading factor TOC_dict (dict): configuration parameters for transformer

Returns
substation supply curve for real-time market clearing

Return type
supply_curve_RT (*curve*)

supply_curve(*Prclear*, *FeederCongCapacity*, *FeederPkDemandCapacity*, *num_samples*, *Q_max*, *maxPuLoading*, *TOC_dict*)

An internal shared function called by substation_supply_curve_RT and substation_supply_curve_DA functions to generate the supply curve when considering the transformer degradation

Parameters

- **Prclear** (*float*) – retail price overted from wholesale price obtained from trial wholesale node clearing, in $/kWh
- **FeederCongCapacity** (*float*) – feeder congestion capacity, in kWh
- **FeederPkDemandCapacity** (*float*) – feeder peak demand, in kWh
- **num_samples** (*int*) – number of sampling points
- **Q_max** (*float*) – substation limit, in kWh
- **maxPuLoading** (*float*) – maximum pu loading factor
- **TOC_dict** (*dict*) – configuration parameters for transformer

Variables:
FeederCongPrice (float): feeder congestion price, in $/kWh FeederPkDemandPrice (float): feeder peak demand price, in $/kWh

Returns
quantity sampling of the supply curve, in kWh SupplyPrices (list): prices sampling of the supply curve, in $/kWh

Return type
SupplyQuantities (list)

test_function()

Test function with the only purpose of returning the name of the object

trial_wholesale_clearing(*curve_ws_node*, *curve_DSO*, *day*, *hour*)

An internal shared function called by set_Pwclear_RT and set_Pwclear_DA functions to implement the trial wholesale node clearing

Parameters

- **curve_ws_node** (Curve) – wholesale node curve
- **curve_DSO** (Curve) – aggregated demand curve at DSO level
- **day** –
- **hour** –

Returns
cleared price, in $/kWh cleared_quantity(float): cleared quantity, in kWh trial_clear_type (int): clear type

Return type
Pwclear (float)

update_wholesale_node_curve()

Update the wholesale node curves according to the most updated curve coefficients, may be updated every day

`tesp_support.dsot.dso_market.test()`

tesp_support.dsot.dso_quadratic_curves module

Class that prepares the quadratic curves for DSO market

class `tesp_support.dsot.dso_quadratic_curves.DSO_LMPs_vs_Q(config_path='LMP_DATA', file_name='/Annual_DA_LMP_Load_data.csv')`

Bases: `object`

This object creats the quadractive curves witht historical data from the base case

Parameters

- **config_path** (*str*) – path to file directory
- **file_name** (*str*) – name of the CSV file containing the historical DA LMP prices with associated quantities

config_path

path to file directory

Type
str

df_dsos_lml_q

list of lmps and associated quantity trou time

Type
list od dataframes

lmps_names

list of lmps names

Type
list of str

q_lmps_names

list of quantities names

Type
list of str

degree

degree of curve to be fitted

Type
int

coeficients_weekday

24 arrays of 3 for every DSO

Type
array of arrays

coeficients_weekend

24 arrays of 3 for every DSO

Type
array of arrays

c_to_DSO_m(*DSO*, *C*)

Convert coefficients to DSO market

Parameters

- **DSO** (*int*) – DSO identifier
- **C** (*int*) – from 0 to 2 especifing the curve coeficient being taken

fit_model(*i*, *p_time*)

Fit a quadractive curve utilizing sklearn

Parameters

- **i** (*int*) – DSO identifier
- **p_time** (*np array bool*) – True for samples utilized in fiiting the quadractic curve

Returns
[['1', 'x', 'x^2']] quadractic curve coeficients

Return type
df_dsos_lml_q (array)

make_json_out()

Save the fitted curve to json

multiple_fit_calls()

Calls the fit model for each scenario

The scenarios are hour of day and day type (i.e., weekday and weekends)

organize_remuve_outliers(*data_da*)

Orginize and remuve outliers from dataframe of multiple DSO

Parameters
data_da (*dataframe*) – contaings historical data from DA LMPs with associated quantities

Returns
every element of the list is a DSO with historical price and quantiti. The index is pandas datetime.

Return type
df_dsos_lml_q (list of dataframes)

tesp_support.dsot.dso_rate_making module

@author: reev057

tesp_support.dsot.dso_rate_making.**DSO_rate_making**(*case*, *dso_num*, *metadata*, *dso_expenses*, *tariff_path*, *dso_scaling_factor*, *num_indust_cust*)

Main function to call for calculating the customer energy consumption, monthly bills, and tariff adjustments to ensure revenue matches expenses. Saves meter and bill dataframes to a hdf5 file. :param case: directory path for the case to be analyzed :type case: str :param dso_num: number of the DSO folder to be opened :type dso_num: str :param metadata: :param dso_expenses: dso expenses that need to be matched by customer revenue

:type dso_expenses: TBD :param tariff_path: :param dso_scaling factor: multiplier on customer bills to reflect the total number of customers in the DSO :type dso_scaling factor: float :param num_indust_cust: number of industrial customers :type num_indust_cust: int

Returns
dataframe of energy consumption and max 15 minute power consumption for each month and total bill_df : dataframe of monthly and total bill for each house broken out by each element (energy, demand, connection, and total bill) tariff: Updated dictionary of tariff structure with rates adjusted to ensure revenue meets expenses surplus: dollar value difference between dso revenue and expenses. When converged should be tiny (e.g. 1e-12)

Return type
meter_df

`tesp_support.dsot.dso_rate_making.`**`annual_energy`**(*month_list*, *folder_prefix*, *dso_num*, *metadata*)

Creates a dataframe of monthly energy consumption values and annual sum based on monthly h5 files. :param month_list: list of lists. Each sub list has month name (str), directory path (str) :type month_list: list :param folder_prefix: prefix of GLD folder name (e.g. '/TE_base_s') :type folder_prefix: str :param dso_num: number of the DSO folder to be opened :type dso_num: str :param metadata: metadata of GridLAB-D model entities :type metadata: dict

Returns
dataframe of energy consumption and max 15 minute power consumption for each month and total year_energysum_df: dataframe of energy consumption summations by customer class (res., commercial, and indust)

Return type
year_meter_df

`tesp_support.dsot.dso_rate_making.`**`calc_cust_bill`**(*metadata*, *meter_df*, *trans_df*, *energy_sum_df*, *tariff*, *dso_num*, *SF*, *ind_cust*)

Calculate the customer bill using summary meter data and fixed tariff structure. :param metadata: metadata structure for the DSO to be analyzed :type metadata: dict :param meter_df: monthly and total energy consumption and peak power by house (meter) :type meter_df: dataframe :param trans_df: :param energy_sum_df: :param tariff: dictionary of fixed tariff structure :type tariff: dict :param dso_num: :param SF: Scaling factor to scale GLD results to TSO scale (e.g. 1743) :type SF: float :param ind_cust: number of industrial customers :type ind_cust: int

Returns
dataframe of monthly and total bill for each house broken out by each element (energy, demand, connection, and total bill)

Return type
bill_df

`tesp_support.dsot.dso_rate_making.`**`get_cust_bill`**(*cust*, *bill_df*, *bill_metadata*)

Populates dictionary of individual customer's annual bill. :param cust: customer name (meter name from GLD dictionary) :type cust: str :param bill_df: dataframe of annual and monthly customer bills :type bill_df: dataframe :param bill_metadata: dictionary of GLD metadata including tarrif and building type for each meter :type bill_metadata: dict

Returns
dictionary of customers annual energy bill

Return type
customer_annual_bill (dict)

`tesp_support.dsot.dso_rate_making.`**`read_meters`**(*metadata*, *dir_path*, *folder_prefix*, *dso_num*, *day_range*, *SF*, *dso_data_path*)

Determines the total energy consumed and max power consumption for all meters within a DSO for a series of days. Also collects information on day ahead and real time quantities consumed by transactive customers. Creates summation of these statistics by customer class. :param metadata: metadata structure for the DSO to be analyzed :type metadata: dict :param dir_path: directory path for the case to be analyzed :type dir_path: str :param folder_prefix: prefix of GLD folder name (e.g. 'TE_base_s') :type folder_prefix: str :param dso_num: number of the DSO folder to be opened :type dso_num: str :param day_range: range of days to be summed (for example a month). :type day_range: list :param SF: Scaling factor to scale GLD results to TSO scale (e.g. 1743) :type SF: float

Returns
dataframe of energy consumption and max 15 minute power consumption for each month and total energysum_df: dataframe of energy consumption summations by customer class (residential, commercial, and industial) saves the two dataframe above to an h5 file in the dir_path

Return type
meter_df

tesp_support.dsot.ev_agent module

Class that controls the Electric Vehicle

Implements the optimum schedule of charging and discharging DA; generate the bids for DA and RT; monitor and supervisory control of GridLAB-D environment element.

The function call order for this agent is:
initialize()

set_price_forecast(forecasted_price)

Repeats at every hour:

- formulate_bid_da() {return BID}
- set_price_forecast(forecasted_price)

Repeats at every 5 min:

- set_battery_SOC(msg_str) {updates C_init}
- formulate_bid_rt() {return BID}
- inform_bid(price) {update RTprice}
- bid_accepted() {update inv_P_setpoint and GridLAB-D P_out if needed}

class `tesp_support.dsot.ev_agent.`**`EVDSOT`**(*ev_dict*, *inv_properties*, *key*, *model_diag_level*, *sim_time*, *solver*)

Bases: `object`

This agent manages the electric vehicle (ev)

Parameters

- **ev_dict** (*dict*) –
- **inv_properties** (*dict*) –
- **key** (*str*) –
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

- **solver** (*str*) –

Initialize from Args

name

name of this agent

Type
str

Rc

rated charging power in kW for the battery

Type
float

Rd

rated discharging power in kW for the battery

Type
float

Lin

battery charging loss in %

Type
float

Lout

battery discharging loss in %

Type
float

Cmin

minimum allowable stored energy in kWh (state of charge lower limit)

Type
float

Cmax

maximum allowable stored energy in kWh (state of charge upper limit)

Type
float

Cinit

initial stored energy in the battery in kWh

Type
float

evCapacity

battery capacity in kWh

Type
float

batteryLifeDegFactor

constant to model battery degradation

Type
float

`windowLength`

length of day ahead optimization period in hours (e.g. 48-hours)

Type
int

`dayAheadCapacity`

% of battery capacity reserved for day ahead bidding

Type
float

`No initialization required`

`bidSpread`

this can be used to spread out bids in multiple hours. When set to 1 hour (recommended), it's effect is none

Type
int

`P`

location of P in bids

Type
int

`Q`

location of Q in bids

Type
int

`f_DA`

forecasted prices in $/kWh for all the hours in the duration of windowLength

Type
float) (1 X windowLength

`ProfitMargin_slope`

specified in % and used to modify slope of bid curve. Set to 0 to disable

Type
float

`ProfitMargin_intercept`

specified in % to generate a small dead band (i.e., change in price does not affect quantity). Set to 0 to disable

Type
float

`pm_hi`

Highest possible profit margin in %

Type
float

pm_lo

Lowest possible profit margin in %

Type
float

RT_state_maintain

true if battery must maintain charging or discharging state for 1 hour

Type
bool

RT_state_maintain_flag

(0) not define at current hour (-1) charging (+1) discharging

Type
int

RT_flag

if True, has to update GridLAB-D

Type
bool

inv_P_setpoint

next GridLAB-D inverter power output

Type
float

optimized_Quantity

Optimized quantity

Type
float) (1 X Window Length

#not used if not biding DA

prev_clr_Quantity

cleared quantities (kWh) from previous market iteration for all hours

Type
float) (1 X Window Length

prev_clr_Price

cleared prices ($/kWh) from previous market iteration

Type
float) (1 X windowLength

BindingObjFunc

if True, then optimization considers cleared price, quantities from previous iteration in the objective function

Type
bool

DA_cleared_price(*price*)

Set the DA_cleared_price attribute

Parameters
price (*float*) – cleared price in $/kWh

DA_model_parameters(*sim_time*)

DA_optimal_quantities()

Generates Day Ahead optimized quantities for EV

Returns
Optimal quantity from optimization for all hours of the window specified by windowLength

Return type
Quantity (float) (1 x windowLength)

RT_fix_four_points_range(*BID*, *Ql*, *Qu*)

Verify feasible range of RT bid

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **Ql** –
- **Qu** –

Returns
4 point bid only the feasible range

Return type
BIDr (float) ((1,2)X4)

RT_gridlabd_set_P(*model_diag_level*, *sim_time*)

Update variables for ev output "inverter"

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

inv_P_setpoint is a float in W

bid_accepted(*current_time*)

Update the P and Q settings if the last bid was accepted

Returns
True if the inverter settings changed, False if not.

Return type
bool

con_rule_eq1(*m*, *i*)

con_rule_eq2(*m*, *i*)

con_rule_eq3(*m*, *i*)

con_rule_eq4(*m*, *i*)

con_rule_ine1(*m*, *i*)

con_rule_ine2(*m*, *i*)

formulate_bid_da()

Formulate 4 points of P and Q bids for the DA market

Function calls "DA_optimal_quantities" to obtain the optimal quantities for the DA market. With the quantities, the 4 point bids are formulated.

Before returning the BID the function resets "RT_state_maintain_flag" which, if RT_state_maintain is TRUE, the battery will be forced to keep its state (i.e., charging or discharging).

Returns
store last DA market bids

Return type
BID (float) (((1,2)X4) X windowLength)

formulate_bid_rt()

Formulates RT bid

Uses the last 4 point bid from DA market and consider current state of charge of the ev. Will change points to change points for feasible range of Qmin Qmax points if necessary. Furthermore, allows a maximum deviation of +/-100% from the DA plan.

Returns
bid in Real Time market

Return type
realTimeBid (float) ((1,2) x 4)

from_P_to_Q_ev(*BID*, *PRICE*)

Convert the 4 point bids to a quantity with the known price

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **PRICE** (*float*) – cleared price in $/kWh

Returns
_quantity -> active power (-) charging (+) discharging

Return type
float

get_car_home_duration(*cur_secs*, *interval*)

Return the duration of car at home during the given 'interval' seconds starting from cur_secs

Parameters

- **cur_secs** – (seconds) current (starting) time with reference of midnight as 0
- **interval** – (seconds) duration in which status needs to be estimated

Returns
duration in seconds for which car is at home in given interval

get_uncntrl_ev_load(*sim_time*)

Function returns 48-hour forecast of ev load in base case w/o optimization

Parameters
sim_time (*datetime*) –

Returns
48-hour forecast of ev load in base case w/o optimization

Return type
list

`inform_bid`(*price*)

Set the cleared_price attribute

Parameters
price (*float*) – cleared price in $/kWh

`is_car_home`(*cur_secs*)

Is the Car is at home

Parameters
cur_secs – current time in seconds

Returns
if car is at home at cur_secs is True or otherwise False

Return type
bool

`is_car_leaving_home`(*cur_secs*, *interval*)

Tells if car is leaving from home during the given 'interval'

Parameters

- **cur_secs** – (seconds) current (starting) time with reference of midnight as 0
- **interval** – (seconds) duration in which status needs to be estimated

Returns

if car is leaving from home during the given 'interval'
seconds starting from cur_secs is True or otherwise False

Return type
bool

`obj_rule`(*m*)

`set_ev_SOC`(*msg_str*, *model_diag_level*, *sim_time*)

Set the ev state of charge

Updates the self.Cinit of the battery

Parameters

- **msg_str** (*str*) – message with ev SOC in percentage
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

`set_price_forecast`(*forecasted_price*)

Set the f_DA attribute

Parameters
forecasted_price (*float x 48*) – cleared price in $/kWh

`test_function`()

Test function with the only purpose of returning the name of the object

`tesp_support.dsot.ev_agent.test()`

Testing

Makes a single battery agent and run DA

tesp_support.dsot.forecasting module

Class responsible for forecasting

Implements the substation level DA price forecast and load forecast

class `tesp_support.dsot.forecasting.Forecasting(port, config_Q)`

Bases: `object`

This Class perform the forecast

Parameters

- **TODO** (#) – update inputs
- **TODO** – Load base case run files

 # TODO

update attributes

add_skew_scalar(*datafr*, *N_skew*, *N_scalar*)

Skew the values with given seconds and multiply by scalar in the whole year dataframe

Args: datafr (DataFrame): dataframe created with the schedule name for a year N_skew (int): number of seconds to skew either (+ or -)

calc_solar_flux(*cpt*, *day_of_yr*, *lat*, *sol_time*, *dnr_i*, *dhr_i*, *vertical_angle*)

calc_solargain(*day_of_yr*, *time*, *dnr*, *dhr*, *lat*, *lon*, *tz_offset*)

correcting_Q_forecast_10_AM(*Q_10_AM*, *offset*, *day_of_week*)

Correcs the quantity submited to the wolsale market at 10 AM

Parameters

Q_10_AM (*list of 24 float*) – DA quantities

Returns

Corrected 10 AM Quantities

forecasting_schedules(*name*, *time*, *len_forecast=48*)

get_internal_gain_forecast(*skew_scalar*, *time*, *extra_forecast_hours=0*)

Forecast the electric zip_load and internal gain of all zip loads of a house by reading schedule files and applying skew. Forecast is for 48-hours ahead from start time :param skew_scalar: dictionary containing 'zip_skew', 'zip_scalar' and 'zip_heatgain_fraction' for each zip load :param 'zip_skew' is a scalar and same for all type of zip loads for the given house. 'zip_scalar' and 'zip_heatgain_fraction': :param are dictionary containing different values for each tyoe of zip load: :param time: Datetime format: forecast start time :param extra_forecast_hours: (int) number of hours for which forecast needs to be stored. For example if it is 24, then :param we need to get forecast for 48+24=72 hours so that there is no need to come back to this function for next 24-hours.:

Returns

list of (48+extra_forecast_hours) values of total zipl loads and total internal gain due to zip loads

get_solar_forecast(*time*, *dso_num*)

get_solar_gain_forecast(*climate_conf*, *current_time*)

get_substation_unresponsive_industrial_load_forecast(*peak_load=3500.0*)

Get substation unresponsive industrial load forecast

Args:
peak_load (float): peak load in kWh

Returns
forecast of next 48-hours unresponsive load

Return type
base_run_load (float x 48)

get_substation_unresponsive_load_forecast(*peak_load=7500.0*)

Get substation unresponsive load forecast

TODO: Update to model that make use of the base case run files TODO: Get weather forecast from weather agent

Parameters
peak_load (*float*) – peak load in kWh

Returns
forecast of next 48-hours unresponsive load

Return type
base_run_load (float x 48)

get_waterdraw_forecast(*skew_scalar*, *time*)

static initialize_schedule_dataframe(*start_time*, *end_time*)

Initialize the data frame for one year

Parameters

- **start_time** (*datetime, str*) – time in str format - DD/MM/YYY HH:MT:SS
- **end_time** (*datetime, str*) – time in str format - DD/MM/YYY HH:MT:SS

make_dataframe_schedule(*filename*, *schedule_name*)

Reads .glm files with multiple schedule names and makes dataframe for a year for given schedule name

Parameters

- **filename** (*str*) – name of glm file to be loaded
- **schedule_name** (*str*) – name of the schedule to be laoded

set_retail_price_forecast(*DA_SW_prices*)

Set substation price forecast

Nonsummable diminishing.

Parameters
DA_SW_prices (*float x 48*) – cleared price in $/kWh from the last shifting window run

Returns
forecasted prices in $/kWh for the next 48-hours

Return type
forecasted_price (float x 48)

`set_sch_year(year)`

`set_solar_diffuse_forecast(fncs_str)`

Set the 48-hour solar diffuse forecast :param param fncs_str: solar_diffuse_forecast ([float x 48]):

`set_solar_direct_forecast(fncs_str)`

Set the 48-hour solar direct forecast :param param fncs_str: solar_direct_forecast ([float x 48]):

`set_temperature_forecast(fncs_str)`

Set the 48-hour temperature forecast

Parameters
fncs_str – temperature_forecast ([float x 48]): predicted temperature in F

`tesp_support.dsot.forecasting.test()`

tesp_support.dsot.gen_map module

`tesp_support.dsot.gen_map.prepare_network(node, node_col, high_renewables_case, zero_pmin=False, zero_index=False, on_ehv=True, split_start_cost=False, high_ramp_rates=False, coal=True)`

tesp_support.dsot.generator_balance_sheet_func module

@author: yint392

`tesp_support.dsot.generator_balance_sheet_func.generator_balance_sheet_annual(generator_num, gen_type, meta_path, system_path, path_to_write, write_to_txt=False, write_to_JSON=False)`

tesp_support.dsot.glm_dictionary module

Functions to create metadata from a GridLAB-D input (GLM) file

Metadata is written to a JSON file, for convenient loading into a Python dictionary. It can be used for agent configuration, e.g., to initialize a forecasting model based on some nominal data. It's also used with metrics output in post-processing.

Public Functions:

glm_dict
Writes the JSON metadata file.

`tesp_support.dsot.glm_dictionary.append_include_file(lines, fname)`

`tesp_support.dsot.glm_dictionary.ercotMeterName(objname)`

Enforces the meter naming convention for ERCOT

Replaces anything after the last _ with *mtr*.

Parameters
objname (*str*) – the GridLAB-D name of a house or inverter

Returns
The GridLAB-D name of upstream meter

Return type
str

`tesp_support.dsot.glm_dictionary.glm_dict(name_root, config=None, ercot=False)`

Writes the JSON metadata file from a GLM file

This function reads *name_root.glm* and writes *[name_root]_glm_dict.json* The GLM file should have some meters and triplex_meters with the bill_mode attribute defined, which identifies them as billing meters that parent houses and inverters. If this is not the case, ERCOT naming rules can be applied to identify billing meters.

Parameters

- **name_root** (*str*) – path and file name of the GLM file, without the extension
- **config** (*dict*) –
- **ercot** (*bool*) – request ERCOT billing meter naming. Defaults to false. — THIS NEEDS TO LEAVE THIS PLACE
- **te30** (*bool*) – request hierarchical meter handling in the 30-house test harness. Defaults to false. — THIS NEEDS TO LEAVE THIS PLACE

`tesp_support.dsot.glm_dictionary.ti_enumeration_string(tok)`

if thermal_integrity_level is an integer, convert to a string for the metadata

tesp_support.dsot.helpers_dsot module

Utility functions for use within tesp_support, including new agents. This is DSO+T specific helper functions

class `tesp_support.dsot.helpers_dsot.Curve(pricecap, num_samples)`

Bases: `object`

Accumulates a set of price, quantity bidding curves for later aggregation

Parameters

- **pricecap** (*float*) – the maximum price that is allowed in the market, in $/kWh
- **num_samples** (*int*) – the number of sampling points, describes how precisely the curve is sampled

prices

array of prices, in $/kWh

Type
[float]

quantities

array of quantities, in kW

Type
[float]

uncontrollable_only

equals to 1 when there is only uncontrollable load demand bids in the market

Type
bool

curve_aggregator(*identity*, *bid_curve*)

Adding one more bid curve to the aggregated seller or buyer curve

Args:
identity (str): identifies whether the bid is collected from a "Buyer" or "Seller" bid_curve ([list]): a nested list with dimension (m, 2), with m equals 2 to 4

curve_aggregator_DSO(*substation_demand_curve*)

Adding one substation bid curve to the aggregated DSO bid curve,
applied when then curve instance is a DSO demand curve

Args:
substation_demand_curve(Curve): a curve object representing the aggregated substation demand curve

update_price_caps()

Update price caps based on the price points

class tesp_support.dsot.helpers_dsot.**HvacMode**(*value*)

Bases: IntEnum

Describes the operating mode of the HVAC

COOLING = 0

HEATING = 1

class tesp_support.dsot.helpers_dsot.**MarketClearingType**(*value*)

Bases: IntEnum

Describes the market clearing type

CONGESTED = 1

FAILURE = 2

UNCONGESTED = 0

tesp_support.dsot.helpers_dsot.**curve_bid_sorting**(*identity*, *bid_curve*)

Sorting the 4-point curve bid primarily on prices and secondarily on quantities

For "Buyer", the bid prices are ordered descendingly and bid quantities are ordered ascendingly; For "Seller", both the bid prices and the bid quantities are ordered descendingly;

Parameters

- **identity** (*str*) – identifies whether the bid is collected from a "Buyer" or "Seller"
- **bid_curve** (*[list]*) – unsorted curve bid

Outputs:
sorted_bid_curve ([list]): sorted curve bid

`tesp_support.dsot.helpers_dsot.get_intersect(a1, a2, b1, b2)`

`tesp_support.dsot.helpers_dsot.resample_curve(x_vec, y_vec, min_q, max_q, num_samples)`

`tesp_support.dsot.helpers_dsot.resample_curve_for_market(x_vec_1, y_vec_1, x_vec_2, y_vec_2)`

`tesp_support.dsot.helpers_dsot.resample_curve_for_price_only(x_vec_1, x_vec_2, y_vec_2)`

`tesp_support.dsot.helpers_dsot.test()`

`tesp_support.dsot.helpers_dsot.write_dsot_management_script(master_file, case_path, system_config=None, substation_config=None, weather_config=None)`

Write experiment management scripts from JSON configuration data, linux and helics only

Reads the simulation configuration file or dictionary and writes

- run.{sh, bat}, simple run script to launch experiment
- kill.{sh, bat}, simple run script to kill experiment
- clean.{sh, bat}, simple run script to clean generated output files from the experiment

Parameters

- **master_file** (*str*) – name of the master file to the experiment case
- **case_path** (*str*) – path to the experiment case
- **system_config** (*dict*) – configuration of the system for the experiment case
- **substation_config** (*dict*) – configuration of the substations in the experiment case
- **weather_config** (*dict*) – configuration of the climates being used

`tesp_support.dsot.helpers_dsot.write_dsot_management_script_f(master_file, case_path, system_config=None, substation_config=None, weather_config=None)`

Write experiment management scripts from JSON configuration data, windows and linux, fncs only

Reads the simulation configuration file or dictionary and writes

- run.{sh, bat}, simple run script to launch experiment
- kill.{sh, bat}, simple run script to kill experiment
- clean.{sh, bat}, simple run script to clean generated output files from the experiment

Parameters

- **master_file** (*str*) – name of the master file to the experiment case
- **case_path** (*str*) – path to the experiment case
- **system_config** (*dict*) – configuration of the system for the experiment case
- **substation_config** (*dict*) – configuration of the substations in the experiment case
- **weather_config** (*dict*) – configuration of the climates being used

`tesp_support.dsot.helpers_dsot.write_management_script(archive_folder, case_path, outPath, gld_Debug, run_post)`

`tesp_support.dsot.helpers_dsot.write_mircogrids_management_script(master_file, case_path, system_config=None, substation_config=None, weather_config=None)`

Write experiment management scripts from JSON configuration data, linux ans helics only

Reads the simulation configuration file or dictionary and writes

- run.{sh, bat}, simple run script to launch experiment
- kill.{sh, bat}, simple run script to kill experiment
- clean.{sh, bat}, simple run script to clean generated output files from the experiment

Parameters

- **master_file** (*str*) – name of the master file to the experiment case
- **case_path** (*str*) – path to the experiment case
- **system_config** (*dict*) – configuration of the system for the experiment case
- **substation_config** (*dict*) – configuration of the substations in the experiment case
- **weather_config** (*dict*) – configuration of the climates being used

`tesp_support.dsot.helpers_dsot.write_players_msg(case_path, sys_config, dt)`

tesp_support.dsot.hvac_agent module

Class that …

TODO: update the purpose of this Agent

class `tesp_support.dsot.hvac_agent.HVACDSOT(hvac_dict, house_properties, key, model_diag_level, sim_time, solver)`

Bases: `object`

This agent …

TODO: update the purpose of this Agent

Parameters

- **hvac_dict** (*dict*) –
- **house_properties** (*dict*) –
- **key** (*str*) –
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable
- **solver** (*str*) –

 # TODO

update attributes for this agent

DA_model_parameters(*moh3*, *hod3*, *dow3*)

self.basepoint_cooling = 73.278 self.temp_min_cool = self.temp_min_cool + self.basepoint_cooling self.temp_max_cool = self.temp_max_cool + self.basepoint_cooling self.thermostat_mode = 'Cooling'

if self.thermostat_mode == 'Cooling':

temp_min_48hour = self.temp_min_cool temp_max_48hour = self.temp_max_cool

else:

temp_min_48hour = self.temp_min_heat temp_max_48hour = self.temp_max_heat

DA_optimal_quantities()

Generates Day Ahead optimized quantities for Water Heater according to the forecasted prices and water draw schedule, called by DA_formulate_bid function

Returns

Optimized quantities for each hour in the DA bidding horizon, in kWh

Return type

Quantity (list) (1 x windowLength)

bid_accepted(*model_diag_level*, *sim_time*)

Update the thermostat setting if the last bid was accepted

The last bid is always "accepted". If it wasn't high enough, then the thermostat could be turned up.

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

Returns

True if the thermostat setting changes, False if not.

Return type

bool

calc_etp_model()

Sets the ETP parameters from configuration data

References

Thermal Integrity Table Inputs and Defaults

calc_solar_flux(*cpt*, *day_of_yr*, *lat*, *sol_time*, *dnr_i*, *dhr_i*, *vertical_angle*)

calc_solargain(*day_of_yr*, *start_hour*, *dnr*, *dhr*, *lat*, *lon*, *tz_offset*)

calc_thermostat_settings(*model_diag_level*, *sim_time*)

Sets the ETP parameters from configuration data

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

References

Table 3 - Easy to use slider settings

change_basepoint(*moh*, *hod*, *dow*, *model_diag_level*, *sim_time*)

Updates the time-scheduled thermostat setting

Parameters

- **moh** (*int*) – the minute of the hour from 0 to 59
- **hod** (*int*) – the hour of the day, from 0 to 23
- **dow** (*int*) – the day of the week, zero being Monday
- **model_diag_level** (*int*) – Specific level for logging errors; set to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

Returns
True if the setting changed, False if not

Return type
bool

change_solargain(*moh*, *hod*, *dow*)

Updates the pre-recorder solar gain

Parameters

- **moh** (*int*) – the minute of the hour from 0 to 59
- **hod** (*int*) – the hour of the day, from 0 to 23
- **dow** (*int*) – the day of the week, zero being Monday

Updates:
solar_gain

con_rule_eq1(*m*, *t*)

formulate_bid_da()

Formulate windowLength hours 4 points PQ bid curves for the DA market

Function calls DA_optimal_quantities to obtain the optimal quantities for the DA market. With the quantities, a 4 point bids are formulated for each hour.

Returns
BID (float) (windowLength X 4 X 2): DA bids to be sent to the retail DA market

formulate_bid_rt(*model_diag_level*, *sim_time*)

Bid to run the air conditioner through the next period for real-time

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

Returns
[bid price $/kwh, bid quantity kW] x 4

Return type
[[float, float], [float, float], [float, float], [float, float]]

get_scheduled_setpt(*moh3*, *hod4*, *dow3*)

Parameters

- **moh3** – (int): the minute of the hour from 0 to 59
- **hod4** – (int): the hour of the day, from 0 to 23
- **dow3** – (int): the day of the week, zero being Monday

get_solargain(*climate_conf*, *current_time*)

estimates the nominal solargain without solargain_factor

Parameters

- **climate_conf** – latitude and longitude info in a dict
- **current_time** – the time for which solargain needs to be estiamted

get_uncntrl_hvac_load(*moh*, *hod*, *dow*)

inform_bid(*price*)

Set the cleared_price attribute

Parameters

price (*float*) – cleared price in $/kwh

obj_rule(*m*)

set_air_temp(*fncs_str*, *model_diag_level*, *sim_time*)

Sets the air_temp attribute

Parameters

- **fncs_str** (*str*) – FNCS message with temperature in degrees Fahrenheit
- **model_diag_level** (*int*) – Specific level for logging errors; set to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

set_da_cleared_quantity(*BID*, *PRICE*)

Convert the 4 point bids to a quantity with the known price

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **PRICE** (*float*) – cleared price in $/kWh

Returns

cleared quantity

Return type

quantity (float)

set_house_load(*fncs_str*)

Sets the hvac_load attribute, if greater than zero

Parameters

fncs_str (*str*) – FNCS message with load in kW

set_humidity(*fncs_str*)

Sets the humidity attribute

Parameters
fncs_str (*str*) – FNCS message with humidity

set_humidity_forecast(*fncs_str*)

Set the 48-hour price forecast and calculate min and max

Parameters
fncs_str – temperature_forecast ([float x 48]): predicted temperature in F

set_hvac_load(*fncs_str*)

Sets the hvac_load attribute, if greater than zero

Parameters
fncs_str (*str*) – FNCS message with load in kW

set_hvac_state(*fncs_str*)

Sets the hvac_on attribute

Parameters
fncs_str (*str*) – FNCS message with state, ON or OFF

set_internalgain_forecast(*internalgain_array*)

Set the 48-hour internalgain forecast :param internalgain_array: internalgain_forecast ([float x 48]): forecasted internalgain in BTu/h

set_price_forecast(*price_forecast*)

Set the 24-hour price forecast and calculate mean and std

Parameters
price_forecast (*[float x 24]*) – predicted price in $/kwh

set_solar_diffuse(*fncs_str*)

Sets the solar diffuse attribute, if greater than zero

Parameters
fncs_str (*str*) – FNCS message with solar irradiance

set_solar_direct(*fncs_str*)

Sets the solar irradiance attribute, if greater than zero

Parameters
fncs_str (*str*) – FNCS message with solar irradiance

set_solargain_forecast(*solargain_array*)

Set the 48-hour solargain forecast

Parameters
solargain_array – solargain_forecast ([float x 48]): forecasted solargain in BTu/(h*sf)

set_temperature(*fncs_str*)

Sets the outside temperature attribute

Parameters
fncs_str (*str*) – FNCS message with outdoor temperature in F

set_temperature_forecast(*fncs_str*)

Set the 48-hour price forecast and calculate min and max

Parameters
fncs_str – temperature_forecast ([float x 48]): predicted temperature in F

set_voltage(*fncs_str*)

Sets the mtr_v attribute

Parameters
fncs_str (*str*) – FNCS message with meter line-neutral voltage

set_wh_load(*fncs_str*)

Sets the wh_load attribute, if greater than zero

Parameters
fncs_str (*str*) – FNCS message with load in kW

set_zipload_forecast(*forecast_ziploads*)

Set the 48-hour zipload forecast :param forecast_ziploads: array of zipload forecast

Returns: nothing, sets the property

store_full_internalgain_forecast(*forecast_internalgain*)

Parameters
forecast_internalgain – internal gain forecast to store for future

Returns: sets the variable so that it can be used later hours as well

store_full_zipload_forecast(*forecast_ziploads*)

Parameters
forecast_ziploads – internal gain forecast to store for future

Returns: sets the variable so that it can be used later hours as well

temp_bound_rule(*m*, *t*)

test_function()

Test function with the only purpose of returning the name of the object

update_temp_limits_da(*cooling_setpt*, *heating_setpt*)

tesp_support.dsot.hvac_agent.**test**()

Testing

Makes a single hvac agent and run DA

tesp_support.dsot.load_less_solar module

Creates a new 200-bus load profile that is the original load profile less the distributed solar generation for that bus. For the hourly profile the solar data that is used is the hourly forecast data. For the five-minute profile the actual 5minute solar data is used. Users can select whether they need to generate hourly or five-minute datasets.

An 8-bus aggregated profile is also created (hourly or five-minute) automatically after the full 200-bus load dataset has been created.

class tesp_support.dsot.load_less_solar.**Mode**(*value*)

Bases: Enum

An enumeration.

FIVE_MINUTE = 1

HOUR = 0

`tesp_support.dsot.load_less_solar.`**`create_8_node_load_less_solar`**(*dso_meta*, *load_dir*, *input_load_filename*, *output_load_filename*)

Using information in the dso_meta dictionary, this function aggregates the 200-bus load-less-solar values into the 8-bus values. The resulting dataset is written out to file.

Parameters

- **`dso_meta`** – DSO metadata defining how 200-node buses map to 8-node buses
- **`load_dir`** – Directory with input load data file
- **`input_load_filename`** – Filename only, no path
- **`output_load_filename`** – Filename only, no path

Returns
(none)

`tesp_support.dsot.load_less_solar.`**`create_load_less_solar`**(*input_load_filename*, *output_load_filename*, *solar_dir*, *load_dir*, *mode*)

Hourly data is used for the DA market and needs to subtract the distributed generation solar forecast. 5-minute data is RT and needs to subtract the actual solar production.

Sample hourly load data:
Hour_End,Bus0,Bus1,Bus2,Bus3,Bus4,Bus5,Bus6,… 12/29/2015 0:00,3659.8,248.49,1652,4415.4,201.28,50.374,83.668,… 12/29/2015 1:00,3659.8,248.49,1652,4415.4,201.28,50.374,83.668,… 12/29/2015 2:00,3603.9,246.54,1626.8,4348,198.2,49.784,82.688,… 12/29/2015 3:00,3577.2,246.19,1614.8,4315.8,196.74,49.431,… 12/29/2015 4:00,3595.8,247.46,1623.1,4338.1,197.76,49.543,… 12/29/2015 5:00,3666,249.03,1654.8,4422.9,201.62,50.085,83.188,… 12/29/2015 6:00,3795.8,253.74,1713.4,4579.4,208.75,51.893,… 12/29/2015 7:00,3931.7,260.87,1774.8,4743.5,216.23,53.583,…

Sample hourly DSO distributed solar forecast data:
0 0 0 0 0 0 0 0 1.079 2.518 3.499 6.463

Sample 5-minute load data:
Seconds,Bus1,Bus2,Bus3,Bus4,Bus5,Bus6,Bus7,Bus8, 0,3659.8,248.5,1652,4415.4,201.3,50.4,83.7,145.1 300,3659.8,248.5,1652,4415.4,201.3,50.4,83.7,145.1 600,3659.8,248.5,1652,4415.4,201.3,50.4,83.7,145.1 900,3659.8,248.5,1652,4415.4,201.3,50.4,83.7,145.1 1200,3659.8,248.5,1652,4415.4,201.3,50.4,83.7,145.1

Sample 5-minute solar data:
12/29/15 0:00,0 12/29/15 0:05,0 12/29/15 0:10,0 12/29/15 0:15,0 12/29/15 0:20,0 12/29/15 0:25,0 12/29/15 0:30,0 12/29/15 0:35,0 12/29/15 0:40,0 12/29/15 0:45,0 12/29/15 0:50,0 12/29/15 0:55,0 12/29/15 1:00,0

Parameters

- **`load_filename`** – Filename of input load data
- **`solar_dir`** – Directory of input solar data
- **`load_dir`** – Directory of input load data

Returns
list of list of load data less solar

Return type
load_data

`tesp_support.dsot.load_less_solar.`**`data`**(*self*, *message*, **args*, ***kws*)

`tesp_support.dsot.load_less_solar.`**`parse_DSO_metadata_Excel`**(*dso_metadata_path_Excel*, *worksheet_name*)

This function parses the DSO metadata which is contained in a JSON and Excel files. Most of the metadata is in the JSON but one crucial piece of information is in the Excel file: the mapping of the 200-node to 8-node buses.

Sample of the bus-generator file: (Note the first columns is empty)

200 bus 8 bus
1 1 2 1 3 1 4 1 5 1 6 1 ...

Parameters

- **dso_metadata_path_Excel** (*str*) –
- **parsed.** (*containing the metadata to be*) –
- **worksheet_name** (*str*) –
- **metadata** (*containing the*) –

Returns
dso_meta (list of dicts) - One dictionary per DSO with appropriate metadata captured.

`tesp_support.dsot.load_less_solar.`**`read_load_file`**(*load_path*)

Parameters
load_path – Path to load file that is being read in

Returns
list of lists with the headers and load data

Return type
load_data

`tesp_support.dsot.load_less_solar.`**`write_out_load_file`**(*load_data*, *out_path*)

Writes out load data to CSV file. Assumes a list of lists format.

Parameters

- **load_data** – Load data to be written to CSV.
- **out_path** – Path and filename of output file

Returns:

tesp_support.dsot.plots module

`tesp_support.dsot.plots.`**`DSO_loadprofiles`**(*dso_num*, *dso_range*, *day_range*, *case*, *dso_metadata_file*, *metadata_path*, *plot_weather=True*)

For a specified dso range and case this function will analyze the ratios of Res, Comm, and Industrial.

Parameters

- **dso_num** (*str*) – number of the DSO folder to be opened
- **dso_range** (*range*) – the DSO range that the data should be analyzed
- **day_range** (*list*) – range of days to be summed (for example a month)
- **case** (*str*) – folder extension of case of interest
- **dso_metadata_file** (*str*) – folder extension and file name for DSO metadata

Returns
dataframe with analysis values saves values to file

`tesp_support.dsot.plots.RCI_analysis(dso_range, case, data_path, metadata_path, dso_metadata_file, energybill=False)`

For a specified dso range and case this function will analyze the ratios of Res, Comm, and Industrial. :param dso_range: the DSO range that the data should be analyzed :type dso_range: list :param case: folder extension of case of interest :type case: str :param data_path: :type data_path: str :param metadata_path: :type metadata_path: str :param dso_metadata_file: DSO and Comm Building metadata :type dso_metadata_file: str :param energybill: :type energybill: bool

Returns
dataframe with analysis values saves values to file

`tesp_support.dsot.plots.TicTocGenerator()`

`tesp_support.dsot.plots.amenity_loss(gld_metadata, dir_path, folder_prefix, dso_num, day_range)`

Determines the loss of amenity metrics (aka unmet hours) for HVAC and WH.

Parameters

- **gld_metadata** (*dict*) – gld metadata structure for the DSO to be analyzed
- **dir_path** (*str*) – directory path for the case to be analyzed
- **folder_prefix** (*str*) – prefix of GLD folder name (e.g. '/TE_base_s')
- **dso_num** (*str*) – number of the DSO folder to be opened
- **day_range** (*range*) – range of days to be summed (for example a month).

Returns
dataframe of loss of amenity metrics for HVAC and WH

Return type
amenity_df

`tesp_support.dsot.plots.annual_amenity(metadata, month_list, folder_prefix, dso_num)`

Creates a dataframe of monthly energy consumption values and annual sum based on monthly h5 files. :param month_list: list of lists. Each sub list has month name (str), directory path (str) :type month_list: list :param folder_prefix: prefix of GLD folder name (e.g. '/TE_base_s') :type folder_prefix: str :param dso_num: number of the DSO folder to be opened :type dso_num: str

Returns
dataframe of energy consumption and max 15 minute power consumption for each month and total year_energysum_df: dataframe of energy consumption summations by customer class (res., commercial, and indust)

Return type
year_meter_df

`tesp_support.dsot.plots.bldg_load_stack(dso, day_range, case, agent_prefix, gld_prefix, metadata_path, daily_dso_plots=False)`

For a specified dso, system, variable, and day this function will load in the required data, plot the daily profile and save the plot to file. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: int :param day_range: range of starting day and ending day of data to include :type day_range: range :param case: folder extension of case of interest :type case: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str :param metadata_path: path of folder containing metadata :type metadata_path: str :param daily_dso_plots: :type daily_dso_plots: bool

Returns
saves daily profile plot to file

`tesp_support.dsot.plots.bldg_stack_plot(dso_range, day_range, case, metadata_path)`

For a specified dso, system, variable, and day this function will load in the required data, plot the daily profile and save the plot to file. :param dso_range: :type dso_range: list :param day_range: range of starting day and ending day of data to include :type day_range: range :param case: folder extension of case of interest :type case: str :param metadata_path: path of folder containing metadata :type metadata_path: str

Returns
saves daily profile plot to file

`tesp_support.dsot.plots.customer_comparative_analysis(case_data, comp_data, case_path, comp_path, dso_num, dso_metadata_path, month='sum', slice=None)`

Creates a comparison of change in energy consumption and anemities for all customers :param case_data: path location for reference case with annual energy and amenity data :type case_data: str :param comp_data: path location for comparison case with annual energy and amenity data :type comp_data: str :param case_path: path location for reference case with simulation metadata for agents/GLD etc. :type case_path: str :param comp_path: path location for comparison case with simulation metadata for agents/GLD etc. :type comp_path: str :param dso_num: dso to be plotted :type dso_num: str :param dso_metadata_path: path to location of DSO metadata files :type dso_metadata_path: str :param month: month of annual analysis to be plotted. set to 'sum' to plot aggregate of all data. :type month: str :param slice: sub set of data to be plotted (e.g. 'residential', 'office', 'HVAC' :type slice: str

Returns
saves plot to file.

`tesp_support.dsot.plots.customer_meta_data(glm_meta, agent_meta, dso_metadata_path)`

Update GLM dictionary with information from agent dictionary needed for customer billing.

Parameters

- **glm_meta** (*dict*) – dictionary of GridLAB-D information
- **agent_meta** (*dict*) – dictionary of transactive agent information
- **dso_metadata_path** (*str*) – location of metadata for commercial buildings

Returns
dictionary of GridLAB-D information

Return type
glm_meta (dict)

`tesp_support.dsot.plots.daily_load_plots(dso, system, subsystem, variable, day, case, comp, agent_prefix, gld_prefix)`

For a specified dso, system, variable, and day this function will load in the required data, plot the daily profile and save the plot to file. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: str :param system: the system to be plotted (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum', 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day: the day to plotted. :type day: str :param case: folder extension of case of interest :type case: str :param comp: folder extension of a comparison case of interest. Optional - set to None to show just one case. :type comp: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
saves daily profile plot to file

`tesp_support.dsot.plots.daily_summary_plots(dso, system, subsystem, variable, day_range, case, comp, oper, diff, denom, agent_prefix, gld_prefix)`

For a specified day range, system, variable, and dso this function will load in the required data and plot the variable for each day based on the operator and compare it to another case (optional). :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: str :param system: the system to be plotted (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum, 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day_range: range of the day indexes to be plotted. Day 1 has an index of 0 :type day_range: range :param case: folder extension of case of interest :type case: str :param comp: folder extension of comparison case of interest :type comp: str :param oper: operator for selecting a scalar value to represent the daily range (e.g. 'min, 'max', 'mean') :type oper: str :param diff: If True will plot the difference between the baseline (case) and comparison (comp) :type diff: bool :param denom: denominator that values should be divided by before plotting :type denom: value :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
saves plots to file

`tesp_support.dsot.plots.der_load_stack(dso, day_range, case, gld_prefix, metadata_path)`

For a specified dso and day range this function will load in the required data, process the data for the stacked DER loads and save the data to file. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: int :param day_range: the day range to plotted. :type day_range: range :param case: folder extension of case of interest :type case: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str :param metadata_path: path of folder containing metadata :type metadata_path: str

Returns
saves dso DER loads data to file

`tesp_support.dsot.plots.der_stack_plot(dso_range, day_range, metadata_path, case, comp=None)`

For a specified dso range and day range this function will load in the required data, plot the stacked DER loads and save the plot to file. :param dso_range: the DSO range that should be plotted. :type dso_range: list :param day_range: the day range to plotted. :type day_range: range :param metadata_path: path of folder containing metadata :type metadata_path: str :param case: folder extension of case of interest :type case: str :param comp: folder extension for reference case to be plotted in comparison :type comp: str

Returns
saves hdf and csv data files of combined dso data saves DER stack plot to file

`tesp_support.dsot.plots.df_reduction(df, subsystem, variable, format)`

This utility slices a dataframe based on the subsystem (or aggregation) of interest. This is used for agent or house data where there is multiple house data per timestep and reduction is needed before plotting data. :param df: dataframe to be reduced :type df: dataframe :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum', 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param format: flag as to whether the data is GridLAB-D ('gld') or agent ('agent') as the format is slightly different. :type format: str

Returns
reduced dataframe

Return type
df (dataframe)

`tesp_support.dsot.plots.dso_comparison_plot(dso_range, system, subsystem, variable, day, case, agent_prefix, gld_prefix)`

For a specified dso range, system, variable, and day this function will load in the required data, plot the variable

for all DSOs and save the plot to file. :param dso_range: the DSO range that the data should be plotted for :type dso_range: range :param system: the system to be plotted (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum, 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day: the day to plotted. :type day: str :param case: folder extension of case of interest :type case: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
saves dso comparison plot to file

`tesp_support.dsot.plots.`**`dso_forecast_stats`**(*dso_range*, *day_range*, *case*, *dso_metadata_file*, *ercot_dir*)

For a specified dso range and day range this function will load in the required data, plot forecast errors for all for DSOs and save the plots to file. :param dso_range: the DSO range that the data should be plotted for :type dso_range: list :param day_range: the day to plotted. :type day_range: range :param case: folder extension of case of interest :type case: str :param dso_metadata_file: path and file name of the dso metadata file :type dso_metadata_file: str :param ercot_dir: path location of the ercot and industial load metadata :type ercot_dir: str

Returns
saves dso comparison plot to file

`tesp_support.dsot.plots.`**`dso_lmp_stats`**(*month_list*, *output_path*, *renew_forecast_file*)

For a specified dso range and list of month path information this function will load in the required data, and summarize DSO LMPs versus loads for all months, and plot comparisons. :param month_list: list of lists. Each sub list has month name (str), directory path (str) :type month_list: list :param output_path: path of the location where output (plots, csv) should be saved :type output_path: str :param renew_forecast_file: path and name of ercot renewable generation forecast csv file :type renew_forecast_file: str

Returns
saves dso load comparison plots to file saves csv of RT and DA loads and LMPs to file

`tesp_support.dsot.plots.`**`dso_load_stats`**(*dso_range*, *month_list*, *data_path*, *metadata_path*, *plot=False*)

For a specified dso range and list of month path information this function will load in the required data, and summarize DSO loads for all months, plot comparisons, and find Qmax. :param dso_range: the DSO range that the data should be plotted for :type dso_range: list :param month_list: list of lists. Each sub list has month name (str), directory path (str) :type month_list: list :param data_path: path of the location where output (plots, csv) should be saved :type data_path: str :param metadata_path: location of ercot load data :type metadata_path: str

Returns
saves dso load comparison plots to file saves summary of Qmax for each DSO to file

`tesp_support.dsot.plots.`**`dso_market_plot`**(*dso_range*, *day*, *case*, *dso_metadata_file*, *ercot_dir*)

For a specified dso range and day this function will load in the required data, plot standard market price and quantity values all for DSOs and save the plots to file. :param dso_range: the DSO range that the data should be plotted for :type dso_range: list :param day: the day to plotted. :type day: str :param case: folder extension of case of interest :type case: str :param dso_metadata_file: path and file name of the dso metadata file :type dso_metadata_file: str :param ercot_dir: path location of the ercot and industial load metadata :type ercot_dir: str

Returns
saves dso comparison plot to file

`tesp_support.dsot.plots.`**`find_edge_cases`**(*dso*, *base_case*, *day_range*, *agent_prefix*, *gld_prefix*)

For a specified dso and case this function will return the day associated with a range of 'edge cases' for example the hottest day or biggest swing in prices. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: str :param base_case: folder extension of case of interest :type base_case: str :param day_range: range of

days to be summed (for example a month) :type day_range: list :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
dataframe with values for each day dictionary of worst values. saves values to file

`tesp_support.dsot.plots.`**`generation_load_profiles`**(*dir_path*, *metadata_path*, *data_path*, *day_range*, *use_ercot_fuel_mix_data=False*, *comp=None*)

For a specified day range this function plots the stacked profiles of all generators (by fuel type) along with load plots. :param dir_path: path locating the AMES data file :type dir_path: str :param metadata_path: path locating the ERCOT load profile 5 minute data :type metadata_path: str :param data_path: path to the folder containing the plots sub folder :type data_path: str :param day_range: range of starting day and ending day of plot :type day_range: range :param use_ercot_fuel_mix_data: If True plots 2016 actual ERCOT data, if False plots AMES RT data. :type use_ercot_fuel_mix_data: bool :param comp: folder path containing the generation data for the comparison case. Set to None for no comparison :type comp: str

Returns
saves plots to file

`tesp_support.dsot.plots.`**`generation_statistics`**(*dir_path*, *config_dir*, *config_file*, *day_range*, *use_gen_data=True*)

For a specified day range this function plots the stacked profiles of all generators (by fuel type) along with load plots. :param dir_path: path locating the AMES data file :type dir_path: str :param config_dir: path locating the case config file :type config_dir: str :param config_file: name of the case config file :type config_file: str :param day_range: range of starting day and ending day of plot :type day_range: range :param use_gen_data: if True uses dispatched generator performance from PyPower. If False uses dispatched :type use_gen_data: bool :param AMES performance:

Returns
saves plots to file

`tesp_support.dsot.plots.`**`get_date`**(*dir_path*, *dso*, *day*)

Utility to return start time (datetime format) of simulation day (str) in question

`tesp_support.dsot.plots.`**`get_day_df`**(*dso*, *system*, *subsystem*, *variable*, *day*, *case*, *agent_prefix*, *gld_prefix*)

This utility loads and returns a dataframe for the desired variable for the day and dso in question. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: str :param system: the system to be plotted (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum, 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day: the day to plotted. :type day: str :param case: folder extension of case of interest :type case: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
reduced dataframe

Return type
df (dataframe)

`tesp_support.dsot.plots.`**`get_house_schedules`**(*agent_metadata*, *gld_metadata*, *house_name*)

Utility to get schedules directly from the agent dictionary. This allows evaluation of schedules prior to agent control. :param agent_metadata: dictionary of agent metadata :type agent_metadata: dict :param gld_metadata: dictionary of gld metadata :type gld_metadata: dict :param house_name: name of GLD house object schedules are wanted for :type house_name: str

Returns
dictionary of schedules in list form

Return type
schedules (dict)

`tesp_support.dsot.plots.heatmap_plots(dso, system, subsystem, variable, day_range, case, agent_prefix, gld_prefix)`

For a specified day, system, variable, and day_range this function will load in the required data, manipulate the dataframe into the required shape, plot the heatmap and save the heatmap to file. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: str :param system: the system to be plotted (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum, 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day_range: range of the day indexes to be plotted. Day 1 has an index of 0 :type day_range: range :param case: folder extension of case of interest :type case: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
saves heatmap to file

`tesp_support.dsot.plots.house_check(dso_range, sourceCase, targetCase, houseProperties)`

Main function of the module.

Parameters
argv – Command line arguments given as: -d <Substation/DSO number to be analyzed> -s <source case folder name> -t <target case folder name> -p <property to compare by plots>

Return type
None

`tesp_support.dsot.plots.limit_check(log_list, dso_num, system, variable, day_range, base_case, agent_prefix, GLD_prefix, max_lim, min_lim)`

For a specified dso, system, variable, and day_range this function will the time and place of the value that most exceeds upper and lower limits. A text description will be added to a log list and returned. :param log_list: list of limit excursions that will be added to. :type log_list: list :param dso_num: the DSO that the data should be plotted for (e.g. '1') :type dso_num: str :param system: the system to be checked (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param variable: variable to be checked from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day_range: range of the day indexes to be checked. Day 1 has an index of 0 :type day_range: range :param base_case: folder extension of case of interest :type base_case: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param GLD_prefix: folder extension for GridLAB-D data :type GLD_prefix: str :param max_lim: upper value that variable should not exceed during simulation :type max_lim: float :param min_lim: lower value that variable should not exceed during simulation :type min_lim: float

Returns
saves heatmap to file

`tesp_support.dsot.plots.load_agent_data(dir_path, folder_prefix, dso_num, day_num, agent_name)`

Utility to open h5 files for agent data.

Parameters

- **dir_path** (*str*) – path of parent directory where DSO folders live
- **folder_prefix** (*str*) – prefix of DSO folder name (e.g. '/TE_base_s')
- **dso_num** (*str*) – number of the DSO folder to be opened
- **day_num** (*str*) – simulation day number (1 = first day of simulation)
- **agent_name** (*str*) – name of agent data to load (e.g. 'house', 'substation', 'inverter' etc)

Returns
dataframe of system metadata agent_df: dataframe of agent timeseries data

Return type
agent_meta_df

`tesp_support.dsot.plots.load_ames_data(dir_path, day_range)`

Utility to open AMES csv file.

Parameters

- **dir_path** (*str*) – path of directory where AMES data lives
- **day_range** (*range*) – range of simulation days for data to be returned

Returns
dataframe of AMES data

Return type
data_df

`tesp_support.dsot.plots.load_da_retail_price(dir_path, folder_prefix, dso_num, day_num, retail=True)`

Utility to return day ahead cleared retail price. Data corresponds to 10am bid day before mapped to actual datetime when the energy will be consumed. :param dir_path: path of parent directory where DSO folders live :type dir_path: str :param folder_prefix: prefix of DSO folder name (e.g. '/TE_base_s') :type folder_prefix: str :param dso_num: number of the DSO folder to be opened :type dso_num: str :param day_num: simulation day number (1 = first day of simulation) :type day_num: str

Returns
dataframe of cleared DA retail price

Return type
retail_da_data_df

`tesp_support.dsot.plots.load_duration_plot(dso, system, subsystem, variable, day, case, comp, agent_prefix, gld_prefix)`

For a specified dso, system, variable, and day this function will load in the required data, plot the load duration profile and save the plot to file. NOTE: currently for one day = should extend to a day-range. :param dso: the DSO that the data should be plotted for (e.g. '1') :type dso: str :param system: the system to be plotted (e.g. 'substation', 'house', 'HVAC_agent') :type system: str :param subsystem: the individual house to be plotted (e.g. 'HousesA_hse_1') or the operator to be used if :type subsystem: str :param aggregating many houses: :type aggregating many houses: e.g. 'sum, 'mean' :param variable: variable to be plotted from system dataframe (e.g. 'cooling_setpoint' or 'real_power_avg') :type variable: str :param day: the day to plotted. :type day: str :param case: folder extension of case of interest :type case: str :param comp: folder extension of a comparison case of interest. Optional - set to None to show just one case. :type comp: str :param agent_prefix: folder extension for agent data :type agent_prefix: str :param gld_prefix: folder extension for GridLAB-D data :type gld_prefix: str

Returns
saves load duration plot to file

`tesp_support.dsot.plots.load_ercot_data(metadata_file, sim_start, day_range)`

Utility to open ercot csv file.

Parameters

- **metadata_file** (*str*) – path of where the metadata_file lives
- **sim_start** (*datetime*) – start time of the simulation (from generate_case_config.json)
- **day_range** (*range*) – range of simulation days for data to be returned

Returns
dataframe of ERCOT 2016 fuel mix data

Return type
data_df

`tesp_support.dsot.plots.load_ercot_fuel_mix(metadata_path, dir_path, day_range)`

Utility to open AMES csv file.

Parameters

- **metadata_path** (*str*) – path of directory where ERCOT Fuel mix data lives
- **dir_path** (*str*) – path of directory where AMES data lives
- **day_range** (*range*) – range of simulation days for data to be returned

Returns
dataframe of AMES data

Return type
data_df

`tesp_support.dsot.plots.load_gen_data(dir_path, gen_name, day_range)`

Utility to open h5 files for agent data.

Parameters

- **dir_path** (*str*) – path of parent directory where DSO folders live
- **gen_name** (*str*) – name of generator (e.g. ‘’)
- **day_range** (*range*) – range of days to be summed (for example a month)

Returns
dataframe of system metadata

Return type
gen_data_df (dataframe)

`tesp_support.dsot.plots.load_indust_data(indust_file, day_range)`

Utility to open industrial load csv file.

Parameters

- **indust_file** (*str*) – path and filename where the industrial csv load lives
- **day_range** (*range*) – range of simulation days for data to be returned

Returns
dataframe of industrial loads per DSO bus

Return type
data_df

`tesp_support.dsot.plots.load_json(dir_path, file_name)`

Utility to open Json files.

`tesp_support.dsot.plots.load_retail_data(dir_path, folder_prefix, dso_num, day_num, agent_name)`

Utility to open h5 files for agent data. :param dir_path: path of parent directory where DSO folders live :type dir_path: str :param folder_prefix: prefix of DSO folder name (e.g. ‘/TE_base_s’) :type folder_prefix: str :param dso_num: number of the DSO folder to be opened :type dso_num: str :param day_num: simulation day number (1 = first day of simulation) :type day_num: str :param agent_name: name of agent data to load (e.g. ‘house’, ‘substation’, ‘inverter’ etc) :type agent_name: str

Returns
dataframe of system metadata agent_df: dataframe of agent timeseries data

Return type
agent_meta_df

`tesp_support.dsot.plots.load_system_data(dir_path, folder_prefix, dso_num, day_num, system_name)`

Utility to open GLD created h5 files for systems' data. :param dir_path: path of parent directory where DSO folders live :type dir_path: str :param folder_prefix: prefix of DSO folder name (e.g. '/TE_base_s') :type folder_prefix: str :param dso_num: number of the DSO folder to be opened :type dso_num: str :param day_num: simulation day number (1 = first day of simulation) :type day_num: str :param system_name: name of system data to load (e.g. 'house', 'substation', 'inverter' etc) :type system_name: str

Returns
dataframe of system metadata system_df: dataframe of system timeseries data

Return type
system_meta_df

`tesp_support.dsot.plots.load_weather_data(dir_path, folder_prefix, dso_num, day_num)`

Utility to open weather dat files and find day of data

Parameters

- **dir_path** (*str*) – path of parent directory where DSO folders live
- **folder_prefix** (*str*) – prefix of DSO folder name (e.g. */DSO_*)
- **dso_num** (*str*) – number of the DSO folder to be opened
- **day_num** (*str*) – simulation day number (1 = first day of simulation)

Returns
dataframe of weather data for simulation day requested

Return type
weather_df

`tesp_support.dsot.plots.metadata_dist_plots(system, sys_class, variable, dso_range, case, data_path, metadata_path, agent_prefix)`

For system, class, and dso_range this function will load in the required data, and plot a histogram of the population distribution. :param system: the system to be plotted (e.g. 'house') :type system: str :param sys_class: the subclass to be plotted (e.g. 'SINGLE_FAMILY'). If system has no subsystems or you want to :type sys_class: str :param see the full population set equal to None: :param variable: variable to be plotted from system dataframe (e.g. 'sqft') :type variable: str :param dso_range: range of the DSOs to be plotted. DSO 1 has an index of 0 :type dso_range: list :param case: folder extension of case of interest :type case: str :param data_path: :type data_path: str :param metadata_path: :type metadata_path: str :param agent_prefix: folder extension for agent data :type agent_prefix: str

Returns
saves plot of population distribution to file

`tesp_support.dsot.plots.plot_lmp_stats(data_path, output_path, dso_num, month_index=8)`

Will plot LMPS by month, duration, and versus netloads loads (for select month), and save to file. :param data_path: location of the data files to be used. :type data_path: str :param output_path: path of the location where output (plots, csv) should be saved :type output_path: str :param dso_num: bus number for LMP data to be plotted :type dso_num: str

Returns
saves dso lmps plots to file

`tesp_support.dsot.plots.run_plots()`

`tesp_support.dsot.plots.tic()`

`tesp_support.dsot.plots.toc(tempBool=True)`

`tesp_support.dsot.plots.transmission_statistics(metadata_file_path, case_config_path, data_path, day_range, sim_results=False)`

For a specified day range this function determines key transmission statistics (e.g. line lenght, max normalized line usage etc).

Parameters

- **metadata_file_path** (*str*) – path and file name of the 8/200-bus metadata json file
- **case_config_path** (*str*) – path and file name locating the system case config json file
- **data_path** (*str*) – path to the folder containing the simulation results
- **day_range** (*range*) – range of starting day and ending day of data to include
- **sim_results** (*bool*) – if True loads in simuation results. If false skips simulation results.

Returns

saves csv statistics to files

`tesp_support.dsot.plots.wind_diff(x)`

tesp_support.dsot.pv_agent module

Class that controls the Photovoltaic Solar agents for now, it only provides day ahead forecast for each agent. It does not participate in bidding

class `tesp_support.dsot.pv_agent.PVDSOT(pv_dict, inv_properties, key, model_diag_level, sim_time)`

Bases: `object`

This agent manages the PV solar

Parameters

- **pv_dict** (*diction*) – dictionary to populate attributes
- **inv_properties** (*diction*) –
- **key** (*str*) – name of this agent
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

Initialize from Args

name

name of this agent

Type

str

participating

participating from pv_dict dictionary

Type

bool

rating

rating from pv_dict dictionary

Type
float

scaling_factor

scaling_factor from pv_dict dictionary

Type
float

slider

slider_setting from pv_dict dictionary

Type
float

windowLength

48 always for now

Type
int

TIME

range(0, self.windowLength)

Type
list

scale_pv_forecast(*solar_f*)

tesp_support.dsot.pv_agent.**test**()

Makes a single pv agent and run DA

tesp_support.dsot.residential_feeder_glm module

tesp_support.dsot.retail_market module

Class that manages the operation of retail market at substation-level

Functionalities include: collecting curve bids from DER and DSO agents; generating aggregated buyer and seller curves; market clearing for both RT and DA retail markets; deciding cleared quantity for individual DERs.

The function call order for this agent is:

initialize(retail_dict)

Repeats at every hour:
clean_bids_DA() curve_aggregator_DA(identity, bid, id) … curve_aggregator_DA(identity, bid, id) clear_market_DA(transformer_degradation, Q_max)

Repeats at every 5 min:
clean_bids_RT() curve_aggregator_RT(identity, bid, id) … curve_aggregator_RT(identity, bid, id) clear_market_RT(transformer_degradation, Q_max)

class tesp_support.dsot.retail_market.**RetailMarket**(*retail_dict*, *key*)

Bases: object

This agent manages the retail market operating

Parameters

- `retail_dict` –
- `key` –

`name`

name of the retail market agent

Type
str

`price_cap`

the maximum price that is allowed in the market, in $/kWh

Type
float

`num_samples`

the number of sampling points, describes how precisely the curve is sampled

Type
int

`windowLength`

length of the planning horizon for the DA market, in hours

Type
int

`Q_max`

capacity of the substation, in kWh

Type
float

`maxPuLoading`

rate of the maxPuLoading for transformer

Type
float

`curve_buyer_RT`

aggregated buyer curve, updated after receiving each RT buyer bid

Type
Curve

`curve_seller_RT`

aggregated seller curve, updated after receiving each RT seller bid

Type
Curve

`curve_buyer_DA`

48 aggregated buyer curves, updated after receiving each DA buyer bid

Type
dict of curves

curve_seller_DA

48 aggregated seller curves, updated after receiving each DA seller bid

Type
dict of curves

clear_type_RT

0=uncongested, 1=congested, 2=inefficient, 3=failure

Type
int

clear_type_DA

list of clear type at each hour

Type
list

cleared_price_RT

cleared price for the substation for the next 5-min

Type
float

cleared_price_DA

list of cleared price at each hour

Type
list

cleared_quantity_RT

cleared quantity for the substation for the next 5-min

Type
float

cleared_quantity_DA

list of cleared quantity at each hour

Type
list

site unresponsive DA

Site Day Ahead quantity which is unresponsive

Type
list

AMES_RT

Smooth Quadratics

Type
list X 5

AMES_DA

X windowLength): Smooth Quadratics

Type
(list X 5

basecase

If true no agent market, false agent market

Type
bool

load_flexibility

If true load is bid in to the market, false all load is unresponsive

Type
bool

U_price_cap_CA

Upper price range for curve aggregator (CA)

Type
float

L_price_cap_CA

Lower price range for curve aggregator (CA)

Type
float

TOC_dict

parameters related to transformer lifetime cost calculation, including OperatingPeriod (int): operating period, in minute timeStep (int): timestep, in minute Tamb (float): ambient temperature, in deg C delta_T_TO_init (int): initial delta temperature of top oil, in deg C delta_T_W_init (int): initial delta temperature of winding, in deg C BP (float): initial cost of transformer, in $ toc_A (float): cost per watt for no-load losses, in $/W toc_B (float): cost per watt for load losses, in $/W Base_Year (float): expected lifespan of transformer, in year P_Rated (float): capacity, in W NLL_rate (float): no load loss rate, in % LL_rate (float): load loss rate, in % Sec_V (float): secondary voltage level, in volt TOU_TOR (float): oil time constant, in minute TOU_GR (float): winding time constant, in minute Oil_n (float): Oil exponent n Wind_m (float): Winding exponent m delta_T_TOR (float): top oil temperature rise, in deg C delta_T_ave_wind_R (float): average winding temperature rise over ambient temperature, in deg C

Type
dict

clean_bids_DA()

Initialize the day-ahead market

clean_bids_RT()

Initialize the real-time market

clear_market(*curve_buyer*, *curve_seller*, *transformer_degradation*, *Q_max*)

Shared function called by both clear_market_RT and clear_market_DA functions to find the intersection between supply curve and demand curve

Parameters

- **curve_buyer** (Curve) – aggregated buyer curve
- **curve_seller** (Curve) – aggregated seller curve
- **transformer_degradation** (*bool*) – equals to 1 if transformer_degradation is considered in the supply curve
- **Q_max** (*float*) – substation capacity, in kWh

Outputs:
clear_type (int) cleared_price (float) cleared_quantity (float) congestion_surcharge (float)

clear_market_DA(*transformer_degradation*, *Q_max*)

Function used for clearing the DA market

Three steps of work are fullfilled in a loop in this function: First the buyer curve at each hour is fitted polynomial; Second clear_market function is called for calculating the cleared price and cleared quantity for the whole market at each hour; Third distribute_cleared_quantity function is called for finding the cleared price and cleared quantity for the individual DERs at each hour.

buyer_info_DA and seller_info_DA, clear_type_DA, cleared_price_DA and cleared_quantity_DA are updated with cleared results

clear_market_RT(*transformer_degradation*, *Q_max*)

Function used for clearing the RT market

Three steps of work are fullfilled in this function: First the buyer curve is fitted polynomial; Second clear_market function is called for calculating the cleared price and cleared quantity for the whole market; Third distribute_cleared_quantity function is called for finding the cleared price and cleared quantity for the individual DERs.

buyer_info_RT and seller_info_RT, clear_type_RT, cleared_price_RT and cleared_quantity_RT are updated with cleared results

convert_2_AMES_quadratic_BID(*curve*, *Q_cleared*, *price_forecast*, *c_type*)

Convert aggregated DSO type bid to AMES quadratic curve

Parameters

- **curve** (Curve) – substation demand curve to be preprocessed
- **price_forecast** – locally forecast price at the substation level
- **Q_cleared** – locally cleared quantity at the substation
- **c_type** (*str*) – 'DA' for day-ahead and 'RT' for real-time

Returns

f(Q)=resp_c2*Q^2+C1*Q+C0
unresp_mw (float): minimum demand MW resp_max_mw (float): maximum demand MW resp_c2 (float): quadratic coefficient resp_c1 (float): linear coefficient resp_c0 (float): constant coefficient resp_deg (int): equal to "2" to represent the current order in the list

Return type
quadratic_bid (list)

curve_aggregator_AMES_DA(*demand_curve_DA*, *Q_max*, *Q_cleared*, *price_forecast*)

Function used to aggregate the substation-level DA demand curves into a DSO-level DA demand curve

Parameters

- **demand_curve_DA** (*dict*) – a collection of demand curves to be aggregated for day-ahead
- **Q_max** (*float*) – maximum capacity of the substation, in kW
- **Q_cleared** (*float*) – locally cleared quantity of the substation in kW
- **price_forecast** (*float*) – locally forecast price at the substation level

curve_aggregator_AMES_RT(*demand_curve_RT*, *Q_max*, *Q_cleared*, *price_forecast*)

Function used to aggregate the substation-level RT demand curves into a DSO-level RT demand curve

Parameters

- **demand_curve_RT** (Curve) – demand curve to be aggregated for real-time
- **Q_max** (*float*) – maximum capacity of the substation, in kW
- **Q_cleared** (*float*) – locally cleared quantity of the substation in kW
- **price_forecast** (*float*) – locally forecast price at the substation level

curve_aggregator_DA(*identity*, *bid_DA*, *name*)

Function used to collect the DA bid and update the accumulated buyer or seller curve

Parameters

- **identity** (*str*) – identifies whether the bid is collected from a "Buyer" or "Seller"
- **bid_DA** (*list*) – a nested list with dimension (self.windowLength, m, 2), with m equals 2 to 4
- **name** (*str*) – name of the buyer or seller

curve_aggregator_RT(*identity*, *bid_RT*, *name*)

Function used to collect the RT bid and update the accumulated buyer or seller curve

Parameters

- **identity** (*str*) – identifies whether the bid is collected from a "Buyer" or "Seller"
- **bid_RT** (*list*) – a nested list with dimension (m, 2), with m equals 2 to 4
- **name** (*str*) – name of the buyer or seller

curve_preprocess(*substation_demand_curve*, *Q_max*)

An internal shared function called by curve_aggregator_DSO_RT and curve_aggregator_DSO_DA functions to truncate

the substation demand curve before aggregation as well as convert the retail prices into wholesale prices

Parameters

- **substation_demand_curve** (Curve) – substation demand curve to be preprocessed
- **Q_max** (*float*) – maximum capacity of the substation, in kW

Returns

preprocessed demand curve

Return type

preprocessed_curve (*curve*)

process_site_da_quantities(*forecast_load*, *name*, *status*)

Function stores the day-ahead quantities, primarily for HVAC at the moment, it utilizes

arguments in: unresponsive loads (1x 48) dataframe responsive loads (1x48) dataframe name: of the house (str) returns self.site_quantity_DA (dict, 'name', ' status (participating or not participating', 'Quantity (1 x 48))

retail_rate_inverse(*Pr*)

Function used to convert the retail prices into wholesale prices

Parameters
Pr (*float*) – retail price, in $/kWh

Returns
wholesale price, in $/kWh

Return type
Pw (float)

test_function()

Test function with the only purpose of returning the name of the object

update_price_CA(*price_forecast*)

Updates the price_CA

tesp_support.dsot.retail_market.**test**()

Testing AMES

tesp_support.dsot.sankey module

tesp_support.dsot.sankey.**label_nodes**(*data*, *total_value*)

Adds max quantity of node value to node label

tesp_support.dsot.sankey.**load_CFS_data**(*results_path*, *dso_range*, *update_data*, *scale*, *labelvals*)

Initiates and updates Sankey diagram data structure for Cash Flow Sheet data. :param results_path: directory path for the case to be analyzed. Should be run after annual post-processing :type results_path: str :param dso_range: range of DSOs to be included in data analysis :type dso_range: range :param update_data: If True pulls in analysis data. If False plots Sankey with default data to show structure :type update_data: bool :param scale: If True scales data to $B from standard $K CFS units :type scale: bool :param labelvals: If True adds quantitative values to node labels :type labelvals: bool

Returns
Sankey data structure for Plotly Sankey diagram plotting

Return type
data (dict)

tesp_support.dsot.sankey.**load_CFS_delta_data**(*results_path*, *comp_path*, *dso_range*, *update_data*, *scale*, *labelvals*, *metadata_file*)

Initiates and updates Sankey diagram data structure for Cash Flow Sheet data. :param results_path: directory path for the case to be analyzed. Should be run after annual post-processing :type results_path: str :param comp_path: directory path for the baseline (business-as-usual) case. :type comp_path: str :param dso_range: range of DSOs to be included in data analysis :type dso_range: range :param update_data: If True pulls in analysis data. If False plots Sankey with default data to show structure :type update_data: bool :param scale: If True scales data to $B from standard $K CFS units :type scale: bool :param labelvals: If True adds quantitative values to node labels :type labelvals: bool :param metadata_file:

Returns
Sankey data structure for Plotly Sankey diagram plotting

Return type
data (dict)

`tesp_support.dsot.sankey.load_energy_data(results_path, dso_range, update_data, scale, labelvals)`

Initiates and updates Sankey diagram data structure for simulation energy data. :param results_path: directory path for the case to be analyzed. Should be run after annual post-processing :type results_path: str :param dso_range: range of DSOs to be included in data analysis :type dso_range: range :param update_data: If True pulls in analysis data. If False plots Sankey with default data to show structure :type update_data: bool :param scale: If True scales data to GW from standard MW CFS units :type scale: bool :param labelvals: If True adds quantitative values to node labels :type labelvals: bool

Returns
Sankey data structure for Plotly Sankey diagram plotting

Return type
data (dict)

`tesp_support.dsot.sankey.rec_diff(d1, d2)`

`tesp_support.dsot.sankey.sankey_plot()`

tesp_support.dsot.solar module

This script provides a few essential functions:

- Download hourly annual solar data for a specified list of locations from the National Solar Radiation Database (NSRDB).
- Simulate the power production for a standard size solar array for each of the solar profiles downloaded using NREL's PySAM (Python wrapper for their SAM tool).
- Aggregate the individual solar profiles to form standardized distributed solar power profiles and centralized utility-scale profiles for the DSO+T 200-node model and 8-node model.

The script includes various housekeeping functions such as file management, automatic generation of distributed solar sites around the primary DSO location, and formatting of the necessary files for ingest by their target simulators.

This script is structured like virtually all of my scripts are structured: running this script with no arguments will perform a comprehensive standard analysis using reasonable default inputs. All necessary inputs and requested outputs will be produced in the "auto-run" folder as a sort of combination example and test case.

Its worth noting that as of this writing, the NSRDB is VERY slow to respond by the provided API and the daily limit of files requests is relatively modest. To gather all the data to run ten distributed solar sites per DSO (200-node) required literally a day.

Also note that use of this API requires registration and use of an NREL- provided API key. This file has my API key removed so that I don't inadvertently share its use with all of PNNL. Get your own key.

DSO+T is simulating 2016 which includes a Leap Day. The NSRDB data does not include data for this day (which is odd, given that the data is based on satellite photos; the source data should exist). I'm replicating data from Feb 28th as the data for Feb. 29th, but I'm only doing this when creating the output files. All the original NRSDB and SAM power profiles only contains 365 days of data.

`tesp_support.dsot.solar.add_locations(dso_meta, solar_meta, nsrdb_path)`

This function adds a number of random locations (specified in solar_meta) withing a certain distance (specified in solar_meta) of a DSO to replicate the effects of the distributed solar generation. Also randomizes the tilt and azimuth orientation of the panels for each distributed location.

The primary location for the DSO is the lat/long from the DSO metadata file and is also added to the solar site list. This is the location that will be used for any utility-scale solar generation.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO
- **solar_meta** (*dict*) – Metadata related to the solar parameters
- **nsrdb_path** (*str*) – Path to the directory with the NSRDB data

Returns
One dictionary per DSO with appropriate metadata captured. Each dictionary has an added item, "distributed locations" which is a dict of parameters for the distributed solar generation

Return type
dso_meta (list of dicts)

`tesp_support.dsot.solar.`**`aggregate_scale_solar_pv_profiles`**(*dso_meta*, *solar_meta*, *output_path*)

This function calculates the utility-scale and distributed solar PV power profiles. The utility scale profile is from a single location (the primary DSO location) while the distributed profile is an equally-weighted average of all profiles for the DSO.

Both the utility-scale and distributed power profiles are scaled to meet both the total solar capacity contribution of the DSO and their relative ratios between utility-scale and dsitributed.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO, pecifically with the load fraction needed to scale the solar PV power profiles as well as the profiles themselves
- **solar_meta** (*dict*) – User-specified solar metadata
- **output_path** (*str*) – Path to the solar PV power profiles for all DSOs.

Returns
List of dicts with metadata associated with each DSO, updated to include the scaled and aggregated utility-scale and dsitributed solar profiles.

Return type
dso_meta (list)

`tesp_support.dsot.solar.`**`aggregate_to_8_nodes`**(*dso_meta*, *output_path*)

This function aggregates the individual profiles for each of the 200 nodes into a single profile for each of the reduced 8 nodes in the smaller model. This is done for both the utility and the distributed solar profiles.

The 8-node DSOs are added to the dso_meta object to the end of the list. They don't have all the same metadata as the original 200-node DSOs.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO, specifically 200-node DSO solar profiles
- **output_path** (*str*) – Location to write out aggregated solar
- **data.** (*profile*) –

Returns
List of dicts with metadata associated with each DSO, updated to include aggregated solar profile data and output file location.

Return type
dso_meta (list)

`tesp_support.dsot.solar.`**`build_dso_solar_folders`**(*dso_meta*, *output_path*)

Creates folders for the solar data in the output path. Folders are only created if they don't already exist.

Parameters

- **dso_meta** – metadata dictionary containing the DSO name
- **output_path** – string specifying output path of data

`tesp_support.dsot.solar.`**`calc_dso_solar_fraction`**(*dso_meta*)

This function calculates the total target PV fraction in proportion with the total average load of the system. That is, it answers the question, "What fraction of the total installed PV should be allocated to each DSO?" (The total size of the installed PV for all of ERCOT is pre-defined by another analysis.)

Parameters
dso_meta (*list*) – List of dicts with metadata associated with each DSO, specifically the average annual load of each DSO

Returns
List of dicts with metadata associated with each DSO, updated to include load fractions for each DSO.

Return type
dso_meta (list)

`tesp_support.dsot.solar.`**`calc_solarPV_power`**(*dso_meta*, *output_path*)

This function uses PySAM from NREL (https://sam.nrel.gov/software-development-kit-sdk/pysam.html) to calculate the solar PV power generation for a solar array at the indicated site. Metadata associated with each site determines the tilt and azimuth. All other array modeling parameters (largely losses) are constant across all sites. The array size is also fixed at a 1 MW.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO, specifically the solar PV information per site
- **output_path** (*str*) – Location to write out solar PV power data

Returns
List of dicts with metadata associated with each DSO, updated to include power profiles for each site in the DSO.

Return type
dso_meta (list)

`tesp_support.dsot.solar.`**`create_GLD_files`**(*dso_meta*)

This function creates 5-minute power profiles from hourly profiles through the power of linear interpolation technology (TM). These profiles are formatted as GridLAB-D tape players.

Example: 2008-12-25 00:00:00,622368.3864 2008-12-25 00:05:00,738071.2498 2008-12-25 00:10:00,680611.3676 2008-12-25 00:15:00,696280.9035

It only acts on the distributed power profiles since these are the ones that are needed by the distribution system simulator to be used to implement distributed (rooftop) generation in the DSO.

Parameters
dso_meta (*list*) – List of dicts with metadata associated with each DSO including the current distributed power profiles.

Returns
List of dicts with DSO data including the location of the interpolated values. Decided not to save these inside the dictionary themselves due to their size and the fact that I don't anticipate needing to reuse them.

Return type
dso-Meta (list)

`tesp_support.dsot.solar.`**`create_dsot_utility_solar_file`**(*dso_meta*, *output_path*)

This function creates a time-series "tape" file for ingest by the DSO+T co-simulation for each of the 200 utility-scale solar PV generation sites. The format is specific to this study (though not opaque in the least); a sample is shown below taken from the wind profile:

time,wind26,wind28,wind29,… 2015-12-29 00:00:00,238.100,646.600,365.400,… 2015-12-29 01:00:00,231.497,646.600,365.400,… …

The data needed for these profiles starts prior to Jan 1 to allow the models to warm up. The data prior to Jan 1 is not crucial, and I'll just the Jan 1 data for those early days as a good-enough approximation.

Also creates an interpolated 5-min version of the file for the RT market.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO including the utility-scale solar PV power profiles.
- **output_path** (*str*) – Path to directory where file will be written

`tesp_support.dsot.solar.`**`create_graphs`**(*dso_meta*, *type*)

This function graphs four weeks out of the year for each DSO to enable easy comparison between the effects of the distributed and utility (single-point) solar power profiles.

Graph image files are saved alongside solar profile files.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO, specifically the distributed and utility scale solar power profiles.
- **type** (*str*) – Keyword indicating the type of graph to create

`tesp_support.dsot.solar.`**`create_hourly_solar_forecast`**(*dso_meta*, *dso_type*, *rng_seed*)

This function creates an hourly forecast for the utility profile by adding noise from a random distribution to it. The parameters for said distribution come from literature (https://www.nrel.gov/docs/fy15osti/63876.pdf).

In a clumsy way that I'm not proud of now that I'm documenting the function, the user decides whether to add noise to the 8-node DSOs or the 200-node DSOs by defining a value for the "dso_type" parameter.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO including the current utility dso power profiles.
- **dso_type** (*int*) – Used to indicate whether to create forecast files for the 8-node or 200-node utility power profiles. Valid values are "8" or "200".

Returns
List of dicts with DSO data including the synthesized forecast profiles.

Return type
dso-Meta (list)

`tesp_support.dsot.solar.`**`download_nsrdb_data`**(*dso_meta*, *solar_meta*, *output_path*)

This function queries the NSRDB database over the web and pulls down the solar data down and stores it in a Pandas dataframe.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO, specifically the site information
- **solar_meta** (*list*) – List of metadata related to the solar parameters

- **output_path** (*str*) – Location to write out solar data from the NSRDB so that we don't have to re-query the database for data we already have.

Returns
List of dicts with metadata associated with each DSO, updated to include NSRDB-specific information such as the abreviated lats and longs from the downloaded solar data as well as a path to each downloaded file and a boolean indicating a file has been downloaded for the given site.

Return type
dso_meta (list)

`tesp_support.dsot.solar.`**`forecast_cleanup`**(*dso_meta*, *idx*, *error*, *profile*, *forecast_profile*)

This function cleans up the forecast power profiles to eliminate negative forecast values and non-zero forecasts when the actual values are zero (used as an indication that it is night). Not a great approximation but good enough.

The 8-node DSOs are added to the dso_meta object to the end of the list. They don't have all the same metadata as the original 200-node DSOs.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO
- **idx** (*int*) – Index for dso_meta to indicate which DSO needs to be cleaned up.
- **error** (*list*) – List of values containing the synthesized error signal
- **forecast_profile** (*list*) – List of values containing the forecast profile with forecast errors added (sometimes causing problems that this function cleans up).

Returns
List of values containing the forecast profile after cleaning.

Return type
forecast_profile (list)

`tesp_support.dsot.solar.`**`generate_KML`**(*dso_meta*, *output_file*)

This function attempts to make a KML file that can be loaded into Google Earth to visualize the solar locations.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO
- **output_file** (*str*) – Location to write output KML

`tesp_support.dsot.solar.`**`generate_forecast_metrics`**(*dso_meta*, *output_path*)

Calculates the RMSE associated with each solar forecast file and write it out to file.

Parameters

- **dso_meta** –
- **output_path** –

Returns
(none)

`tesp_support.dsot.solar.`**`log_metdata`**(*dso_meta*)

`tesp_support.dsot.solar.`**`parse_DSO_metadata_Excel`**(*dso_metadata_path*, *worksheet_name*)

(DEPRECATED): Metadata is now stored in JSON file. See parse_DSO_metadata_Excel_JSON. This function parses the DSO metadata which is contained in an Excel spreadsheet

Parameters

- **dso_metadata_path** (*str*) – Path to the Excel file containing the metadata to be parsed.
- **worksheet_name** (*str*) – Name of the worksheet in the Excel file containing the metadata

Returns
One dictionary per DSO with appropriate metadata captured.

Return type
dso_meta (list of dicts)

`tesp_support.dsot.solar.`**parse_DSO_metadata_Excel_JSON**(*dso_metadata_path_Excel*, *worksheet_name*, *dso_metadata_path_JSON*)

This function parses the DSO metadata which is contained in a JSON and Excel files. Most of the metadata is in the JSON but one crucial piece of information is in the Excel file: the mapping of the 200-node to 8-node buses.

Sample of the bus-generator file: (Note the first columns is empty) 200 bus 8 bus 1 1 2 1 3 1 4 1 5 1 6 1 …

Sample of JSON structure: "general": { … } "DSO_1": { "bus_number": 1, "latitude": 33.02, "longitude": -96.85, "average_load_MW": 4154.30537, … }, …

Parameters

- **dso_metadata_path_Excel** (*str*) – Path to the Excel file containing the metadata to be parsed.
- **worksheet_name** (*str*) – Name of the worksheet in the Excel file containing the metadata.
- **dso_metadata_path_JSON** (*str*) – Path to the JSON file containing the metadata to be parsed.

Returns
One dictionary per DSO with appropriate metadata captured.

Return type
dso_meta (list of dicts)

`tesp_support.dsot.solar.`**parse_solar_metadata**(*solar_metadata_path*)

This function parses the solar metadata JSON.

Parameters
solar_metadata_path (*str*) – Path to the JSON file containing the metadata to be parsed.

Returns
Dictionary form of the data in the JSON file

Return type
solar_dict (dicts)

`tesp_support.dsot.solar.`**truncate**(*f*, *n=3*)

This function truncates the passed in value to the specified number of decimals. The default value is three decimals.

Parameters

- **f** (*string/float*) – Value to be truncated
- **n** (*int*) – Number of decimal places to truncate to

Returns
Truncated value of f

Return type
truc_val (string/float)

`tesp_support.dsot.solar.write_power_profile(output_path, power_data)`

This function writes out a power profile to the provided file. Only intended to write out a single value per line. I probably should have used the standard CSV library instead.

Parameters

- **power_data** (*list*) – List of power values
- **output_path** (*str*) – Path to the location of file to write.

tesp_support.dsot.substation module

Manages the Transactive Control scheme for DSO+T implementation version 1

Public Functions:

dso_loop
initializes and runs the agents

`tesp_support.dsot.substation.dso_loop(metrics_root, with_market)`

Wrapper for *inner_substation_loop*

When *inner_substation_loop* finishes, timing and memory metrics will be printed for non-Windows platforms.

tesp_support.dsot.substation_f module

Manages the Transactive Control scheme for DSO+T implementation version 1

Public Functions:

dso_loop_f
initializes and runs the agents

`tesp_support.dsot.substation_f.dso_loop_f(configfile, metrics_root, with_market)`

Wrapper for *inner_substation_loop*

When *inner_substation_loop* finishes, timing and memory metrics will be printed for non-Windows platforms.

tesp_support.dsot.water_heater_agent module

Class that controls the Water Heater DER

Implements the optimum schedule of heating element operation given DA price forecast; generate the bids for DA and RT; monitor and supervisory control of GridLAB-D environment element with the implementation of new SOHC model and new delta_SOHC model

The function call order for this agent is:

- initialize

Repeats at every hour:

- DA_forecasted_price(forecasted_price)
- DA_forecasted_schedule(forecasted_schedule)
- formulate_bid_da()

Repeats at every 5 min:

- formulate_bid_rt()
- bid_accepted()

class tesp_support.dsot.water_heater_agent.**WaterHeaterDSOT**(*wh_dict*, *wh_properties*, *key*, *model_diag_level*, *sim_time*, *solver*)

Bases: object

This agent manage the operation of water heater

Parameters

- **wh_dict** (*dict*) –
- **wh_properties** (*dict*) –
- **key** (*str*) –
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable
- **solver** (*str*) –

volume

volume of the tank, in gal

Type
float

diameter

diameter of the tank layer, in ft

Type
float

Phw

rated power of the heating elements, in kW

Type
float

Tcold

temperature of the inlet cold water, in degF

Type
float

Tambient

ambient temperature, in degF

Type
float

Tdesired

setpoint value with highest user comfort, in degF

Type
float

Tmax

highest tolerant temperature of the water, in degF

Type
float

`Tmin`

lowest tolerant temperature of the water, in degF

Type
float

`windowLength`

length of DA bidding timeframe

Type
int

`weight_SOHC`

weight of the upper temperature measure when estimating the SOHC, falls into range [0, 1]

Type
float

`weight_comfort`

weight of the user comfort in the DA quantity optimization objective, falls into range [0, 1]

Type
float

`ProfitMargin_intercept`

specified in % and used to modify slope of bid curve. Set to 0 to disable

Type
float

`ProfitMargin_slope`

specified in % to generate a small dead band (i.e., change in price does not affect quantity). Set to 0 to disable

Type
float

`Participating`

equals to 1 when participate in the price-responsive biddings

Type
bool

`price_cap`

the maximum price that is allowed in the retail market, in $/kWh

Type
float

`H_tank`

height of the water tank, in ft

Type
float

`A_tank`

area of the layer of tank, in ft2

Type
float

`A_wall`
area of the water tank wall, in ft2

Type
float

`R_tank`
tank insulation, in ft2*hr*degF/Btu

Type
float

`Cp`
specific heat of water, in Btu/lbm*degF

Type
float

`Rho`
density of the water, in lbm/ft3

Type
float

`BTUperkWh`
unit conversion from kWh to BTU, in BTU/kWh

Type
float

`GALperFt3`
unit conversion from ft3 to gallon, in ga/ft3

Type
float

`T_upper`
current set point of the upper heating element, in degF

Type
float

`T_bottom`
current set point of the bottom heating element, in degF

Type
float

`SOHC`
statue of heat charge, in %

Type
float

`SOHC_desired`
desired SOHC, in %

Type
float

SOHC_max

maximum SOHC, in %

Type
float

SOHC_min

minimum SOHC, in %

Type
float

states_upper

list of states and time in 5-min of upper element

Type
list

states_bottom

list of states and time in 5-min of bottom element

Type
list

runtime_upper

runtime of the upper element during 5-min

Type
float

runtime_bottom

runtime of the lower element during 5-min

Type
float

E_upper

energy consumed by the upper element in 5min, in kWh

Type
float

E_bottom

energy consumed by the bottom elemnt in 5min, in kWh

Type
float

wd_rate

averaged water draw flow rate in the 5min, in gal/min

Type
float

Setpoint_upper

setpoint to be set for the upper element, in degF

Type
float

Setpoint_bottom

setpoint to be set for the bottom element, in degF

Type
float

length_memory

length of memory for the historical data

Type
int

his_T_upper

historical time series data of the temperature measurement at the upper position

Type
list

his_T_bottom

historical time series data of the temperature measurement at the bottom position

Type
list

his_SOHC

historical time series data of the SOHC

Type
list

his_E_upper

historical time series power consumption of upper element

Type
list

his_E_bottom

historical time series power consumption of bottom element

Type
list

his_wd_rate

historical time series water draw flow rate data

Type
list

f_DA_price

forecasted DA price

Type
list

f_DA_schedule

forecasted DA water draw schedule

Type
list

P

index of price in the bid curve matrix

Type
int

Q

index of quantity in the bid curve matrix

Type
int

DA_cleared_prices

list of 48-hours day-ahead cleared prices

Type
list

DA_cleared_quantities

list of 48-hours day-ahead cleared quantities

Type
list

RT_cleared_price

cleared price for the next 5min

Type
float

RT_cleared_quantity

cleared quantity for the next 5min

Type
float

hourto5min

conversion from hour to 5min, equals to 12

Type
int

hour

current hour

Type
int

minute

current minute

Type
int

co0_hour

intercept of the hourly delta SOHC model

Type
float

co1_hour

coefficient of the water draw flow rate in the hourly delta SOHC model

Type
float

co2_hour

coefficient of the upper element consumption in the hourly delta SOHC model

Type
float

co3_hour

coefficient of the bottom element consumption in the hourly delta SOHC model

Type
float

co0_5min

intercept of the 5min delta SOHC model

Type
float

co1_5min

coefficient of the water draw flow rate in the 5min delta SOHC model

Type
float

co2_5min

coefficient of the upper element consumption in the 5min delta SOHC model

Type
float

co3_5min

coefficient of the bottom element consumption in the 5min delta SOHC model

Type
float

RT_SOHC_max

the maximum SOHC the water heater can achieve in the next 5min, in %

Type
float

RT_SOHC_min

the minimum SOHC the water heater can achieve in the next 5min, in %

Type
float

RT_Q_max

higher quantity boundary of the RT bid curve, in kWh

Type
float

RT_Q_min

lower quantity boundary of the RT bid curve, in kWh

Type
float

DA_optimal_quantities()

Generates Day Ahead optimized quantities for Water Heater according to the forecasted prices and water draw schedule, called by DA_formulate_bid function

Returns
Optimized quantities for each hour in the DA bidding horizon, in kWh

Return type
Quantity (list) (1 x windowLength)

bid_accepted(*model_diag_level*, *sim_time*)

Update the thermostat setting if the last bid was accepted

The last bid is always "accepted". If it wasn't high enough, then the thermostat could be turned up.

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

Returns
True if the thermostat setting changes, False if not.

Return type
bool

con_rule_eq1(*m*, *t*)

con_rule_ine1(*m*, *t*)

delta_SOHC_model_5min()

Function used to fit the 5min delta_SOHC estimation model, where 5min delta_SOHC is assumed to be a function of wd_rate E_upper and E_bottom values with 5min interval

delta_SOHC_model_hour()

Function used to fit the hourly delta_SOHC estimation model, where hourly delta_SOHC is assumed to be a function of wd_rate E_upper and E_bottom values with one-hour interval

estimate_wd_rate_5min()

Function used to estimate the water_draw flow rate in the previous 5 mins, called by update_WH_his every 5 mins

formulate_bid_da()

Formulate windowLength hours 4 points PQ bid curves for the DA market

Function calls DA_optimal_quantities to obtain the optimal quantities for the DA market. With the quantities, a 4 point bids are formulated for each hour.

Returns BID (float) (windowLength X 4 X 2): DA bids to be send to the retail DA market

formulate_bid_rt(*model_diag_level*, *sim_time*)

Formulate 4 points PQ bid curve for the RT market

Given the physical and operational constraints of the water heater and the current water heater status, 4 points RT bid curve is formulated for the next 5min.

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors: set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

Returns
RT bid to be send to the retail RT market

Return type
BID (float) (4 X 2)

from_P_to_Q_WH(*BID*, *PRICE*)

Convert the 4 point bids to a quantity with the known price

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **PRICE** (*float*) – cleared price in $/kWh

Returns
quantity to be consumed in the next 5-min

Return type
quantity (float)

get_uncntrl_wh_load()

This simulates the waterheater model without

Returns
48-hours forecast of non transactive waterheater kw consumption without optimization (agent participation)

Return type
list

inform_bid_da(*DAprices*)

Updated the DA_cleared_prices and DA_cleared_quantities attributes when informed by retail market agent

Parameters
DAprices (*list*) – cleared prices from the last DA market clearing, in $/kWh, provided by retail market agent

inform_bid_rt(*RTprice*)

Updated the RT_cleared_prices and RT_cleared_quantities attributes when informed by retail market agent

Parameters
RTprice (*float*) – cleared price from the last RT market clearing, in $/kWh, provided by retail market agent

obj_rule(*m*)

set_air_temp(*fncs_str*, *model_diag_level*, *sim_time*)

Sets the air_temp attribute

Parameters

- **fncs_str** (*str*) – FNCS message with temperature in degrees Fahrenheit
- **model_diag_level** (*int*) – Specific level for logging errors; set to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

set_da_cleared_quantity(*BID*, *PRICE*)

Convert the 4 point bids to a quantity with the known price

Parameters

- **BID** (*float) ((1,2)X4*) – 4 point bid
- **PRICE** (*float*) – cleared price in $/kWh

Returns
cleared quantity

Return type
quantity (float)

set_forecasted_schedule(*forecasted_waterdraw_array*)

Set the f_DA_schedule attribute

Parameters
forecasted_waterdraw_array (*list*) – forecasted waterdraw flow rate schedule in gallons/min, provided by forecast agent

set_price_forecast(*forecasted_price*)

Set the f_DA_price attribute

Parameters
forecasted_price (*list*) – forecasted DA prices in $/kwh, provided by retail market agent

set_time(*hour*, *minute*)

Sets the current hour and minute

Parameters

- **hour** (*int*) – current hour
- **minute** (*int*) – current minute

set_wh_load(*fncs_str*)

Sets the water heater load attribute, if greater than zero

Parameters
fncs_str (*str*) – FNCS message with load in kW

set_wh_lower_state(*fncs_str*)

Sets the lower element state attribute

Parameters
fncs_str (*str*) – FNCS message with ON/OFF status

set_wh_lower_temperature(*fncs_str*, *model_diag_level*, *sim_time*)

Sets the lower tank temperature attribute

Parameters

- **fncs_str** (*str*) – FNCS message with temperature in degrees Fahrenheit
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

set_wh_upper_state(*fncs_str*)

Sets the upper element state attribute

Parameters

fncs_str (*str*) – FNCS message with ON/OFF status

set_wh_upper_temperature(*fncs_str*, *model_diag_level*, *sim_time*)

Sets the upper tank temperature attribute

Parameters

- **fncs_str** (*str*) – FNCS message with temperature in degrees Fahrenheit
- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

set_wh_wd_rate_val(*fncs_str*)

Sets the water draw rate attribute

Parameters

fncs_str (*str*) – FNCS message with wdrate value in gpm

test_function()

Test function with the only purpose of returning the name of the object

update_WH_his(*model_diag_level*, *sim_time*)

Update the historical memory of water heater based on updated readings, called by formulate_bid_rt every 5 mins

Parameters

- **model_diag_level** (*int*) – Specific level for logging errors; set it to 11
- **sim_time** (*str*) – Current time in the simulation; should be human-readable

tesp_support.dsot.water_heater_agent.**test**()

tesp_support.dsot.wind_gen_year module

Written by Ankit Singhal and Mitch Pelton This scripts use Tom's wind model to generate wind power generation and write in csv files. This script should be kept in tesp_support folder

It writes two files:

1. wind.csv: can be considered as actual wind value. It is written with 5 minute resolution. Tom's model generates hourly data which is interpolated to 5 minute (300 seconds) resolution.

2. wind_forecast.csv: a gaussian distribution of error is added to generate hourly wind forecast. First error is added in 5 minute resolution data which then averaged to hourly data.

tesp_support.dsot.wind_gen_year.**generate_wind_data_24hr**(*wind_plants*)

tesp_support.dsot.wind_gen_year.**make_wind_plants**(*ppc*)

tesp_support.dsot.wind_gen_year.**test**()

tesp_support.matpower package

Support files for using MATPOWER - also see instructions on readthedocs for building the FNCS wrapper around the MATLAB run-time

Submodules

tesp_support.matpower.matpower_dict module

`tesp_support.matpower.matpower_dict.`**`matpower_dict`**(*name_root*)

tesp_support.matpower.process_matpower module

`tesp_support.matpower.process_matpower.`**`process_matpower`**(*name_root*)

tesp_support.original package

Submodules

tesp_support.original.case_merge module

Combines GridLAB-D and agent files to run a multi-feeder TESP simulation

Public Functions:

merge_glm
combines GridLAB-D input files

merge_glm_dict
combines GridLAB-D metadata files

merge_agent_dict
combines the substation agent configuration files

merge_substation_yaml
combines the substation agent FNCS publish/subscribe files

merge_fncs_config
combines GridLAB-D FNCS publish/subscribe files

merge_gld_msg
combines GridLAB-D HELICS publish/subscribe configurations

merge_substation_msg
combines the substation agent HELICS publish/subscribe configurations

`tesp_support.original.case_merge.`**`key_present`**(*val*, *ary*)

`tesp_support.original.case_merge.`**`merge_agent_dict`**(*target*, *sources*, *xfmva*)

Combines the substation agent configuration files into target/target.json. The source files must already exist.

Parameters

- **target** (*str*) – the directory and root case name
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.original.case_merge.merge_fncs_config(target, sources)`

Combines GridLAB-D input files into target/target.txt. The source feeders must already exist.

Parameters

- **target** (*str*) – the directory and root case name
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.original.case_merge.merge_gld_msg(target, sources)`

`tesp_support.original.case_merge.merge_glm(target, sources, xfmva)`

Combines GridLAB-D input files into target/target.glm. The source files must already exist.

Parameters

- **target** (*str*) – the directory and root case name
- **sources** (*list*) – list of feeder names in the target directory to merge

`tesp_support.original.case_merge.merge_glm_dict(target, sources, xfmva)`

Combines GridLAB-D metadata files into target/target.json. The source files must already exist.

Each constituent feeder has a new ID constructed from the NamePrefix + original base_feeder, then every child object on that feeder will have its feeder_id, originally network_node, changed to match the new one.

Parameters

- **target** (*str*) – the directory and root case name
- **sources** (*list*) – list of feeder names in the target directory to merge
- **xfmva** (*int*) –

`tesp_support.original.case_merge.merge_substation_msg(target, sources)`

`tesp_support.original.case_merge.merge_substation_yaml(target, sources)`

Combines GridLAB-D input files into target/target.yaml. The source files must already exist.

Parameters

- **target** (*str*) – the directory and root case name
- **sources** (*list*) – list of feeder names in the target directory to merge

tesp_support.original.commercial_feeder_glm module

tesp_support.original.copperplate_feeder_glm module

tesp_support.original.curve module

Utility functions for use within tesp_support, including new agents.

class `tesp_support.original.curve.ClearingType(value)`

Bases: `IntEnum`

Describes the market clearing type

BUYER = 5

EXACT = 3

FAILURE = 1

NULL = 0

PRICE = 2

SELLER = 4

`tesp_support.original.curve.`**`aggregate_bid`**(*crv*)

Aggregates the buyer curve into a quadratic or straight-line fit with zero intercept

Parameters
crv (curve) – the accumulated buyer bids

Returns
Qunresp, Qmaxresp, degree, c2 and c1 scaled to MW instead of kW. c0 is always zero.

Return type
[float, float, int, float, float]

class `tesp_support.original.curve.`**`curve`**

Bases: `object`

Accumulates a set of price, quantity bids for later aggregation. The default order is descending by price.

price

array of prices, in $/kWh

Type
[float]

quantity

array of quantities, in kW

Type
[float]

count

the number of collected bids

Type
int

total

the total kW bidding

Type
float

total_on

the total kW bidding that are currently on

Type
float

total_off

the total kW bidding that are currently off

Type
float

add_to_curve(*price*, *quantity*, *is_on*)

Add one point to the curve

Parameters

- **price** (*float*) – the bid price, should be $/kWhr
- **quantity** (*float*) – the bid quantity, should be kW
- **is_on** (*bool*) – True if the load is currently on, False if not

set_curve_order(*flag*)

Set the curve order (by price) to ascending or descending

Parameters

flag (*str*) – 'ascending' or 'descending'

tesp_support.original.fncs module

Functions that provide access from Python to the FNCS library

Notes

Depending on the operating system, libfncs.dylib, libfncs.dll or libfncs.so must already be installed. Besides the defined Python wrapper functions, these pass-through library calls are always needed:

- *fncs.finalize*: call after the simulation completes
- *fncs.time_request (long)*: request the next time step; blocks execution of this process until FNCS grants the requested time. Then, the process should check for messages from FNCS.

These pass-through calls are also available, but not used in TESP:

- *fncs.route*
- *fncs.update_time_delta*
- *fncs.get_id*
- *fncs.get_simulator_count*
- *fncs.get_events_size*
- *fncs.get_keys_size*
- *fncs.die*: stops FNCS and sends 'die' to other simulators

References

ctypes

FNCS

Examples

- under tesp_support, see substation.py, precool.py and tso_PYPOWER_f.py
- under examples, see loadshed/loadshed.py

`tesp_support.original.fncs.agentGetEvents()`

Retrieve FNCS agent messages

Returns
concatenation of agent messages

Return type
str

`tesp_support.original.fncs.agentPublish(value)`

Publish a value over FNCS, under the configured simulator name / agent name

Parameters
value (*str*) – value

`tesp_support.original.fncs.agentRegister(config=None)`

Initialize the FNCS configuration for the agent interface

Parameters
config (*str*) – a ZPL file. If None (default), provide YAML file in FNCS_CONFIG_FILE environment variable.

`tesp_support.original.fncs.die()`

Call FNCS die because of simulator error

`tesp_support.original.fncs.finalize()`

Call FNCS finalize to end connection with broker

`tesp_support.original.fncs.get_event_at(i)`

Retrieve FNCS message by index number after time_request returns

Returns
one decoded FNCS event

Return type
str

`tesp_support.original.fncs.get_events()`

Retrieve FNCS messages after time_request returns

Returns
tuple of decoded FNCS events

Return type
list

`tesp_support.original.fncs.get_events_size()`

Get the size of the event queue

`tesp_support.original.fncs.get_id()`

Find the FNCS ID

`tesp_support.original.fncs.get_key_at(i)`

Get the topic by index number

Parameters
i (*int*) – the index number

Returns
decoded topic name

Return type
str

`tesp_support.original.fncs.get_keys()`

Find the list of topics

Returns
decoded topic names

Return type
[str]

`tesp_support.original.fncs.get_keys_size()`

Get the size of the keys

`tesp_support.original.fncs.get_name()`

Find the FNCS simulator name

Returns
the name of this simulator as provided in the ZPL or YAML file

Return type
str

`tesp_support.original.fncs.get_simulator_count()`

Find the FNCS simulator count

`tesp_support.original.fncs.get_value(key)`

Extract value from a FNCS message

Parameters
key (*str*) – the topic

Returns
decoded value

Return type
str

`tesp_support.original.fncs.get_value_at(key, i)`

For list publications, get the value by index

Parameters

- **key** (*str*) – the topic
- **i** (*int*) – the list index number

Returns
decoded value

Return type
str

`tesp_support.original.fncs.get_values(key)`

For list publications, get the list of values

Parameters
key (*str*) – the topic

Returns
decoded values

Return type
[str]

`tesp_support.original.fncs.get_values_size(key)`

For list publications, find how many values were published

Parameters
key (*str*) – the topic

Returns
the number of values for this topic

Return type
int

`tesp_support.original.fncs.get_version()`

Find the FNCS version

Returns
major, minor and patch numbers

Return type
int, int, int

`tesp_support.original.fncs.initialize(config=None)`

Initialize the FNCS configuration

Parameters
config (*str*) – a ZPL file. If None (default), provide YAML file in FNCS_CONFIG_FILE environment variable.

`tesp_support.original.fncs.is_initialized()`

Determine whether the FNCS library has been initialized

Returns
True if initialized, False if not.

Return type
bool

`tesp_support.original.fncs.publish(key, value)`

Publish a value over FNCS, under the simulator name

Parameters

- **key** (*str*) – topic under the simulator name
- **value** (*str*) – value

`tesp_support.original.fncs.publish_anon(key, value)`

Publish a value over FNCS, under the 'anonymous' simulator name

Parameters

- **key** (*str*) – topic under 'anonymous'
- **value** (*str*) – value

`tesp_support.original.fncs.route(sender, receiver, key, value)`

Route a value over FNCS from sender to receiver

Parameters

- **sender** (*str*) – simulator routing the message
- **receiver** (*str*) – simulator to route the message to
- **key** (*str*) – topic under the simulator name
- **value** (*str*) – value

`tesp_support.original.fncs.time_request(time)`

FNCS time request

Parameters

time (*int*) – requested time.

`tesp_support.original.fncs.update_time_delta(delta)`

Update simulator time delta value

Parameters

delta (*int*) – time delta.

tesp_support.original.glm_dictionary module

Functions to create metadata from a GridLAB-D input (GLM) file

Metadata is written to a JSON file, for convenient loading into a Python dictionary. It can be used for agent configuration, e.g., to initialize a forecasting model based on some nominal data. It's also used with metrics output in post-processing.

Public Functions:

glm_dict

Writes the JSON metadata file.

`tesp_support.original.glm_dictionary.append_include_file(lines, fname)`

Parameters

- **lines** (*list<str>*) –
- **fname** (*string*) –

`tesp_support.original.glm_dictionary.ercotMeterName(objname)`

Enforces the meter naming convention for ERCOT

Replaces anything after the last _ with *mtr*.

Parameters

objname (*str*) – the GridLAB-D name of a house or inverter

Returns

The GridLAB-D name of upstream meter

Return type

str

`tesp_support.original.glm_dictionary.glm_dict(name_root, ercot=False, te30=False)`

Writes the JSON metadata file from a GLM file

This function reads *name_root.glm* and writes *[name_root]_glm_dict.json* The GLM file should have some meters and triplex_meters with the bill_mode attribute defined, which identifies them as billing meters that parent houses and inverters. If this is not the case, ERCOT naming rules can be applied to identify billing meters.

Parameters

- **name_root** (*str*) – path and file name of the GLM file, without the extension
- **ercot** (*bool*) – request ERCOT billing meter naming. Defaults to false.
- **te30** (*bool*) – request hierarchical meter handling in the 30-house test harness. Defaults to false.

`tesp_support.original.glm_dictionary.isCommercialHouse(house_class)`

`tesp_support.original.glm_dictionary.ti_enumeration_string(tok)`

if thermal_integrity_level is an integer, convert to a string for the metadata

tesp_support.original.hvac_agent module

Class that controls the responsive thermostat for one house.

Implements the ramp bidding method, with HVAC power as the bid quantity, and thermostat setting changes as the response mechanism.

class `tesp_support.original.hvac_agent.hvac(hvac_dict, key, aucObj)`

Bases: `object`

This agent manages thermostat setpoint and bidding for a house

Parameters

- **hvac_dict** (*dict*) – dictionary row for this agent from the JSON configuration file
- **key** (*str*) – name of this agent, also key for its dictionary row
- **aucObj** (`simple_auction`) – the auction this agent bids into

name

name of this agent

Type
str

control_mode

control mode from dict (CN_RAMP or CN_NONE, which still implements the setpoint schedule)

Type
str

houseName

name of the corresponding house in GridLAB-D, from dict

Type
str

meterName

name of the corresponding triplex_meter in GridLAB-D, from dict

Type
str

period

market clearing period, in seconds, from dict

Type
float

wakeup_start

hour of the day (0..24) for scheduled weekday wakeup period thermostat setpoint, from dict

Type
float

daylight_start

hour of the day (0..24) for scheduled weekday daytime period thermostat setpoint, from dict

Type
float

evening_start

hour of the day (0..24) for scheduled weekday evening (return home) period thermostat setpoint, from dict

Type
float

night_start

hour of the day (0..24) for scheduled weekday nighttime period thermostat setpoint, from dict

Type
float

wakeup_set

preferred thermostat setpoint for the weekday wakeup period, in deg F, from dict

Type
float

daylight_set

preferred thermostat setpoint for the weekday daytime period, in deg F, from dict

Type
float

evening_set

preferred thermostat setpoint for the weekday evening (return home) period, in deg F, from dict

Type
float

night_set

preferred thermostat setpoint for the weekday nighttime period, in deg F, from dict

Type
float

weekend_day_start

hour of the day (0..24) for scheduled weekend daytime period thermostat setpoint, from dict

Type
float

weekend_day_set

preferred thermostat setpoint for the weekend daytime period, in deg F, from dict

Type
float

weekend_night_start

hour of the day (0..24) for scheduled weekend nighttime period thermostat setpoint, from dict

Type
float

weekend_night_set

preferred thermostat setpoint for the weekend nighttime period, in deg F, from dict

Type
float

deadband

thermostat deadband in deg F, invariant, from dict

Type
float

offset_limit

maximum allowed change from the time-scheduled setpoint, in deg F, from dict

Type
float

ramp

bidding ramp denominator in multiples of the price standard deviation, from dict

Type
float

price_cap

the highest allowed bid price in $/kwh, from dict

Type
float

bid_delay

from dict, not implemented

Type
float

use_predictive_bidding

from dict, not implemented

Type
float

std_dev

standard deviation of expected price, determines the bidding ramp slope, initialized from aucObj

Type
float

mean

mean of the expected price, determines the bidding ramp origin, initialized from aucObj

Type
float

Trange

the allowed range of setpoint variation, bracketing the preferred time-scheduled setpoint

Type
float

air_temp

current air temperature of the house in deg F

Type
float

hvac_kw

most recent non-zero HVAC power in kW, this will be the bid quantity

Type
float

mtr_v

current line-neutral voltage at the triplex meter

Type
float

hvac_on

True if the house HVAC is currently running

Type
bool

basepoint

the preferred time-scheduled thermostat setpoint in deg F

Type
float

setpoint

the thermostat setpoint, including price response, in deg F

Type
float

bid_price

the current bid price in $/kwh

Type
float

cleared_price

the cleared market price in $/kwh

Type
float

bid_accepted()

Update the thermostat setting if the last bid was accepted

The last bid is always "accepted". If it wasn't high enough, then the thermostat could be turned up.p

Returns
True if the thermostat setting changes, False if not.

Return type
bool

change_basepoint(*hod*, *dow*)

Updates the time-scheduled thermostat setting

Parameters

- **hod** (*float*) – the hour of the day, from 0 to 24
- **dow** (*int*) – the day of the week, zero being Monday

Returns
True if the setting changed, False if not

Return type
bool

formulate_bid()

Bid to run the air conditioner through the next period

Returns
bid price in $/kwh, bid quantity in kW and current HVAC on state, or None if not bidding

Return type
[float, float, bool]

inform_bid(*price*)

Set the cleared_price attribute

Parameters
price (*float*) – cleared price in $/kwh

set_air_temp_from_fncs_str(*val*)

Sets the air_temp attribute

Parameters
val (*str*) – FNCS message with temperature in degrees Fahrenheit

set_air_temp_from_helics(*val*)

set_hvac_load_from_fncs_str(*val*)

Sets the hvac_load attribute, if greater than zero

Parameters
val (*str*) – FNCS message with load in kW

set_hvac_load_from_helics(*val*)

set_hvac_state_from_fncs_str(*val*)

Sets the hvac_on attribute

Parameters

val (*str*) – FNCS message with state, ON or OFF

set_hvac_state_from_helics(*val*)

set_voltage_from_fncs_str(*val*)

Sets the mtr_v attribute

Parameters

val (*str*) – FNCS message with meter line-neutral voltage

set_voltage_from_helics(*val*)

tesp_support.original.parse_msout module

tesp_support.original.parse_msout.**next_matrix**(*fp*, *var*)

tesp_support.original.parse_msout.**next_val**(*fp*, *var*, *bInteger=True*)

tesp_support.original.parse_msout.**read_most_solution**(*fname='msout.txt'*)

tesp_support.original.player_f module

tesp_support.original.player_f.**load_player_loop_f**(*casename*, *keyName*)

tesp_support.original.precool module

Classes for NIST TE Challenge 2 example

The precool_loop class manages time stepping and FNCS messages for the precooler agents, which adjust thermostat set points in response to time-of-use rates and over voltages. The precooler agents also estimate house equivalent thermal parameter (ETP) models based on total floor area, number of stories, number of exterior doors and estimated thermal integrity level. This ETP estimate serves as an example for other agent developers; it's not actually used by the precooler agent.

Public Functions:

precooler_loop

Initializes and runs the precooler agents.

tesp_support.original.precool.**fncs_precool_loop**(*nhours*, *metrics_root*, *dict_root*, *response*)

Function that supervises FNCS messages and time stepping for precooler agents

Opens metrics_root_agent_dict.json and metrics_root_glm_dict.json for configuration. Writes precool_metrics_root.json at completion.

Parameters

- **nhours** (*float*) – number of hours to simulate
- **metrics_root** (*str*) – name of the case, without file extension
- **dict_root** (*str*) – repeat metrics_root, or the name of a shared case dictionary without file extension

- **response** (*str*) – combination of Price and/or Voltage

`tesp_support.original.precool.helics_precool_loop(nhours, metrics_root, dict_root, response, helicsConfig)`

Function that supervises FNCS messages and time stepping for precooler agents

Opens metrics_root_agent_dict.json and metrics_root_glm_dict.json for configuration. Writes precool_metrics_root.json at completion.

Parameters

- **nhours** (*float*) – number of hours to simulate
- **metrics_root** (*str*) – name of the case, without file extension
- **dict_root** (*str*) – repeat metrics_root, or the name of a shared case dictionary without file extension
- **response** (*str*) – combination of Price and/or Voltage
- **helicsConfig** (*str*) – name for HELICS message file

`tesp_support.original.precool.precool_loop(nhours, metrics_root, dict_root, response='PriceVoltage', helicsConfig=None)`

Wrapper for *inner_substation_loop*

When *inner_substation_loop* finishes, timing and memory metrics will be printed for non-Windows platforms.

`class tesp_support.original.precool.precooler(name, agentrow, gldrow, k, mean, stddev, lockout_time, precooling_quiet, precooling_off, bPrice, bVoltage)`

Bases: `object`

This agent manages the house thermostat for time-of-use and overvoltage responses.

References

NIST TE Modeling and Simulation Challenge

Parameters

- **name** (*str*) – name of this agent
- **agentrow** (*dict*) – row from the FNCS configuration dictionary for this agent
- **gldrow** (*dict*) – row from the GridLAB-D metadata dictionary for this agent's house
- **k** (*float*) – bidding function denominator, in multiples of stddev
- **mean** (*float*) – mean of the price
- **stddev** (*float*) – standard deviation of the price
- **lockout_time** (*float*) – time in seconds between allowed changes due to voltage
- **precooling_quiet** (*float*) – time of day in seconds when precooling is allowed
- **precooling_off** (*float*) – time of day in seconds when overvoltage precooling is always turned off

name

name of this agent

Type
str

meterName

name of the corresponding triplex_meter in GridLAB-D, from agentrow

Type
str

night_set

preferred thermostat setpoint during nighttime hours, deg F, from agentrow

Type
float

day_set

preferred thermostat setpoint during daytime hours, deg F, from agentrow

Type
float

day_start_hour

hour of the day when daytime thermostat setting period begins, from agentrow

Type
float

day_end_hour

hour of the day when daytime thermostat setting period ends, from agentrow

Type
float

deadband

thermostat deadband in deg F, invariant, from agentrow, from agentrow

Type
float

vthresh

meter line-to-neutral voltage that triggers precooling, from agentrow

Type
float

toffset

temperature setpoint change for precooling, in deg F, from agentrow

Type
float

k

bidding function denominator, in multiples of stddev

Type
float

mean

mean of the price

Type
float

stddev

standard deviation of the price

Type
float

lockout_time

time in seconds between allowed changes due to voltage

Type
float

precooling_quiet

time of day in seconds when precooling is allowed

Type
float

precooling_off

time of day in seconds when overvoltage precooling is always turned off

Type
float

air_temp

current air temperature of the house in deg F

Type
float

mtr_v

current line-neutral voltage at the triplex meter

Type
float

basepoint

the preferred time-scheduled thermostat setpoint in deg F

Type
float

setpoint

the thermostat setpoint, including price response, in deg F

Type
float

lastchange

time of day in seconds when the setpoint was last changed

Type
float

precooling

True if the house is precooling, False if not

Type
bool

ti

thermal integrity level, as enumerated for GridLAB-D, from gldrow

Type
int

sqft (float

total floor area in square feet, from gldrow

stories

number of stories, from gldrow

Type
int

doors

number of exterior doors, from gldrow

Type
int

UA

heat loss coefficient

Type
float

CA

total air thermal mass

Type
float

HM

interior mass surface conductance

Type
float

CM

total house thermal mass

Type
float

check_setpoint_change(*hour_of_day*, *price*, *time_seconds*)

Update the setpoint for time of day and price

Parameters

- **hour_of_day** (*float*) – the current time of day, 0..24
- **price** (*float*) – the current price in $/kwh
- **time_seconds** (*long long*) – the current FNCS time in seconds

Returns
True if the setpoint changed, False if not

Return type
bool

get_temperature_deviation()

For metrics, find the difference between air temperature and time-scheduled (preferred) setpoint

Returns
absolute value of deviation

Return type
float

make_etp_model()

Sets the ETP parameters from configuration data

References

Thermal Integrity Table Inputs and Defaults

set_air_temp(*val*)

Set the air_temp member variable

Parameters
val (*str*) – FNCS/HELICS message with temperature in degrees Fahrenheit

set_voltage(*val*)

Sets the mtr_v attribute

Parameters
val (*str*) – HELICS message with meter line-neutral voltage

set_voltage_f(*val*)

Sets the mtr_v attribute

Parameters
val (*str*) – FNCS message with meter line-neutral voltage

tesp_support.original.prep_eplus module

tesp_support.original.prep_eplus.**configure_eplus**(*caseConfig*, *template_dir*)

tesp_support.original.prep_eplus.**make_gld_eplus_case**(*fname*, *bGlmReady=False*)

tesp_support.original.prep_eplus.**prepare_bldg_dict**(*caseConfig*)

tesp_support.original.prep_eplus.**prepare_glm_dict**(*caseConfig*)

tesp_support.original.prep_eplus.**prepare_glm_file**(*caseConfig*)

tesp_support.original.prep_eplus.**prepare_glm_helics**(*caseConfig*, *fedMeters*, *fedLoadNames*)

tesp_support.original.prep_eplus.**prepare_run_script**(*caseConfig*, *fedMeters*)

tesp_support.original.prep_eplus.**writeGlmClass**(*theseLines*, *thisClass*, *op*)

tesp_support.original.prep_precool module

Writes the precooling agent and GridLAB-D metadata for NIST TE Challenge 2 example

Public Functions:

prep_precool
writes the JSON and YAML files

`tesp_support.original.prep_precool.`**`prep_precool`**(*name_root*, *time_step=15*)

Sets up agent configurations for the NIST TE Challenge 2 example

Reads the GridLAB-D data from name_root.glm; it should contain houses with thermal_integrity_level attributes. Writes:

- *[name_root]_agent_dict.json*, contains configuration data for the precooler agents
- *[name_root]_precool.yaml*, contains FNCS subscriptions for the precooler agents
- *[name_root]_gridlabd.txt*, a GridLAB-D include file with FNCS publications and subscriptions

Parameters

- **name_root** (*str*) – the name of the GridLAB-D file, without extension
- **time_step** (*int*) – time step period

tesp_support.original.prep_substation module

Sets up the FNCS and agent configurations for te30 and sgip1 examples

This works for other TESP cases that have one GridLAB-D file, one EnergyPlus model, and one PYPOWER model. Use *tesp_case* or *tesp_config* modules to specify supplemental configuration data for these TESP cases, to be provided as the optional *jsonfile* argument to *prep_substation*.

Public Functions:

prep_substation
processes a GridLAB-D file for one substation and one or more feeders

`tesp_support.original.prep_substation.`**`ProcessGLM`**(*fileroot*)

Helper function that processes one GridLAB-D file

Reads fileroot.glm and writes:

- *[fileroot]_agent_dict.json*, contains configuration data for the simple_auction and hvac agents
- *[fileroot]_substation.yaml*, contains FNCS subscriptions for the psimple_auction and hvac agents
- *[fileroot]_gridlabd.txt*, a GridLAB-D include file with FNCS publications and subscriptions
- *[fileroot]_substation.json*, contains HELICS subscriptions for the psimple_auction and hvac agents
- *[fileroot]_gridlabd.json*, a GridLAB-D include file with HELICS publications and subscriptions

Parameters
fileroot (*str*) – path to and base file name for the GridLAB-D file, without an extension

`tesp_support.original.prep_substation.prep_substation(gldfileroot, jsonfile='', bus_id=None)`

Process a base GridLAB-D file with supplemental JSON configuration data

If provided, this function also reads jsonfile as created by *tesp_config* and used by *tesp_case*. This supplemental data includes time-scheduled thermostat set points (NB: do not use the scheduled set point feature within GridLAB-D, as the first FNCS messages will erase those schedules during simulation). The supplemental data also includes time step and market period, the load scaling factor to PYPOWER, ramp bidding function parameters and the EnergyPlus connection point. If not provided, the default values from te30 and sgip1 examples will be used.

Parameters

- **gldfileroot** (*str*) – path to and base file name for the GridLAB-D file, without an extension
- **jsonfile** (*str*) – fully qualified path to an optional JSON configuration file (if not provided, an E+ connection to Eplus_load will be created)
- **bus_id** – substation bus identifier

tesp_support.original.process_agents module

Functions to plot data from GridLAB-D substation agents

Public Functions:

process_agents
Reads the data and metadata, then makes the plots.

`tesp_support.original.process_agents.plot_agents(diction, save_file=None, save_only=False)`

`tesp_support.original.process_agents.process_agents(name_root, diction_name='', save_file=None, save_only=False, print_dictionary=False)`

Plots cleared price, plus bids from the first HVAC controller

This function reads *auction_[name_root]_metrics.json* and *controller_[name_root]_metrics.json* for the data; it reads *[name_root]_glm_dict.json* for the metadata. These must all exist in the current working directory. Makes one graph with 2 subplots:

1. Cleared price from the only auction, and bid price from the first controller
2. Bid quantity from the first controller

Parameters

- **name_root** (*str*) – name of the TESP case, not necessarily the same as the GLM case, without the extension
- **diction_name** (*str*) – metafile name (with json extension) for a different GLM dictionary, if it's not *[name_root]_glm_dict.json*. Defaults to empty.
- **save_file** (*str*) – name of a file to save plot, should include the *png* or *pdf* extension to determine type.
- **save_only** (*bool*) – set True with *save_file* to skip the display of the plot. Otherwise, script waits for user keypress.
- **print_dictionary** (*bool*) – set True to print dictionary.

`tesp_support.original.process_agents.read_agent_metrics(path, name_root, diction_name='', print_dictionary=False)`

tesp_support.original.residential_feeder_glm module

tesp_support.original.simple_auction module

Double-auction mechanism for the 5-minute markets in te30 and sgip1 examples

The substation_loop module manages one instance of this class per GridLAB-D substation.

Todo:

- Initialize and update price history statistics
- Allow for adjustment of clearing_scalar
- Handle negative price bids from HVAC agents, currently they are discarded
- Distribute marginal quantities and fractions; these are not currently applied to HVACs

---

2021-10-29 TDH: Key assumptions we need to refactor out of this: - This is a retail market working under a wholesale market. We want something more general that can operate at any market level. - Load state is not necessary information to run a market. PNNL TSP has traditionally had the construct of responsive and unresponsive loads and I think the idea is crucial for being able to get good quadratic curve fits on the demand curve (by lopping off the unresponsive portion) but I think we should refactor this so that these assumptions are not so tightly integrated with the formulation.

`class tesp_support.original.simple_auction.simple_auction(diction, key)`

Bases: `object`

This class implements a simplified version of the double-auction market embedded in GridLAB-D.

References

Market Module Overview - Auction

Parameters

- **diction** (*diction*) – a row from the agent configuration JSON file
- **key** (*str*) – the name of this agent, which is the market key from the agent configuration JSON file

name

the name of this auction, also the market key from the configuration JSON file

Type
str

std_dev

the historical standard deviation of the price, in $/kwh, from diction

Type
float

mean

the historical mean price in $/kwh, from diction

Type
float

price_cap

the maximum allowed market clearing price, in $/kwh, from diction

Type
float

max_capacity_reference_bid_quantity

this market's maximum capacity, likely defined by a physical limitation in the circuit(s) being managed.

Type
float

statistic_mode

always 1, not used, from diction

Type
int

stat_mode

always ST_CURR, not used, from diction

Type
str

stat_interval

always 86400 seconds, for one day, not used, from diction

Type
str

stat_type

always mean and standard deviation, not used, from diction

Type
str

stat_value

always zero, not used, from diction

Type
str

curve_buyer

data structure to accumulate buyer bids

Type
curve

curve_seller

data structure to accumulate seller bids

Type
curve

refload

the latest substation load from GridLAB-D. This is initially assumed to be all unresponsive and using the load state parameter when adding demand bids (which are generally price_responsive) all loads that bid and are on are removed from the assumed unresponsive load value

Type
float

lmp

the latest locational marginal price from the bulk system market

Type
float

unresp

unresponsive load, i.e., total substation load less the bidding, running HVACs

Type
float

agg_unresp

aggregated unresponsive load, i.e., total substation load less the bidding, running HVACs

Type
float

agg_resp_max

total load of the bidding HVACs

Type
float

agg_deg

degree of the aggregate bid curve polynomial, should be 0 (zero or one bids), 1 (2 bids) or 2 (more bids)

Type
int

agg_c2

second-order coefficient of the aggregate bid curve

Type
float

agg_c1

first-order coefficient of the aggregate bid curve

Type
float

clearing_type

describes the solution type or boundary case for the latest market clearing

Type
ClearingType

clearing_quantity

quantity at the last market clearing

Type
float

clearing_price

price at the last market clearing

Type
float

marginal_quantity

quantity of a partially accepted bid

Type
float

marginal_frac

fraction of the bid quantity accepted from a marginal buyer or seller

Type
float

clearing_scalar

used for interpolation at boundary cases, always 0.5

Type
float

add_unresponsive_load(*quantity*)

aggregate_bids()

Aggregates the unresponsive load and responsive load bids for submission to the bulk system market

clear_bids()

Re-initializes curve_buyer and curve_seller, sets the unresponsive load estimate to the total substation load.

clear_market(*tnext_clear=0*, *time_granted=0*)

Solves for the market clearing price and quantity

Uses the current contents of curve_seller and curve_buyer. Updates clearing_price, clearing_quantity, clearing_type, marginal_quantity and marginal_frac.

Parameters

- **tnext_clear** (*int*) – next clearing time in seconds, should be <= time_granted, for the log file only
- **time_granted** (*int*) – the current time in seconds, for the log file only

collect_bid(*bid*)

Gather HVAC bids into curve_buyer

Also adjusts the unresponsive load estimate, by subtracting the HVAC power if the HVAC is on.

Parameters
bid (*[float, float, bool]*) – price in $/kwh, quantity in kW and the HVAC on state

initAuction()

Sets the clearing_price and lmp to the mean price

2021-10-29 TDH: TODO - Any reason we can't put this in constructor?

set_lmp(*lmp*)

Sets the lmp attribute

Parameters
lmp (*float*) – locational marginal price from the bulk system market

set_refload(*kw*)

Sets the refload attribute

Parameters
kw (*float*) – GridLAB-D substation load in kw

supplier_bid(*bid*)

Gather supplier bids into curve_seller

Use this to enter curves in step-wise blocks.

Parameters
bid (*[float, float]*) – price in $/kwh, quantity in kW

surplusCalculation(*tnext_clear=0*, *time_granted=0*)

Calculates consumer surplus (and its average) and supplier surplus.

This function goes through all the bids higher than clearing price from buyers to calculate consumer surplus, and also accumulates the quantities that will be cleared while doing so. Of the cleared quantities, the quantity for unresponsive loads are also collected. Then go through each seller to calculate supplier surplus. Part of the supplier surplus corresponds to unresponsive load are excluded and calculated separately.

Parameters

- **tnext_clear** (*int*) – next clearing time in seconds, should be <= time_granted, for the log file only
- **time_granted** (*int*) – the current time in seconds, for the log file only

update_statistics()

Update price history statistics - not implemented

tesp_support.original.substation_f module

Manages the simple_auction and hvac agents for the te30 and sgip1 examples

Public Functions:

substation_loop_f
initializes and runs the agents

Todo:

- Getting an overflow error when killing process - investigate whether that happens if simulation runs to completion
- Allow changes in the starting date and time; now it's always midnight on July 1, 2013
- Allow multiple markets per substation, e.g., 5-minute and day-ahead for the DSO+T study

---

tesp_support.original.tesp_case module

tesp_support.original.tesp_config module

tesp_support.original.tesp_monitor module

Presents a GUI to launch a TESP simulation and monitor its progress

Public Functions:

show_tesp_monitor
Initializes and runs the monitor GUI

References

Graphical User Interfaces with Tk

Matplotlib Animation

class tesp_support.original.tesp_monitor.**TespMonitorGUI**(*master*, *HELICS=True*)

Bases: `object`

Manages a GUI with 4 plotted variables, and buttons to stop TESP

The GUI reads a JSON file with scripted shell commands to launch other HELICS/FNCS federates, and a YAML file with HELICS/FNCS subscriptions to update the solution status. Both JSON and YAML files are written by *tesp.tesp_config* The plotted variables provide a sign-of-life and sign-of-stability indication for each of the major federates in the te30 or sgip1 examples, namely GridLAB-D, PYPOWER, EnergyPlus, and the substation_loop that manages a simple_auction with multiple hvac agents. If a solution appears to be unstable or must be stopped for any other reason, exiting the solution monitor will do so.

The plots are created and updated with animated and bit-blitted Matplotlib graphs hosted on a TkInter GUI. When the JSON and YAML files are loaded, the x axis is laid out to match the total TESP simulation time range.

Parameters
master –

root

the TCL Tk toolkit instance

Type
Tk

top

the top-level TCL Tk Window

Type
Window

labelvar

used to display the monitor JSON configuration file path

Type
StringVar

hrs

x-axis data array for time in hours, shared by all plots

Type
[float]

y0

y-axis data array for PYPOWER bus voltage

Type
[float]

y1

y-axis data array for EnergyPlus load

Type
[float]

y2lmp

y-axis data array for PYPOWER LMP

Type
[float]

y2auc

y-axis data array for simple_auction cleared_price

Type
[float]

y3fncs

y-axis data array for GridLAB-D load via FNCS

Type
[float]

y3gld

y-axis data array for sample-and-hold GridLAB-D load

Type
[float]

gld_load

the most recent load published by GridLAB-D; due to the deadband, this value isn't necessary published at every FNCS time step

Type
float

y0min

the first y axis minimum value

Type
float

y0max

the first y axis maximum value

Type
float

y1min

the second y axis minimum value

Type
float

y1max
the second y axis maximum value

Type
float

y2min
the third y axis minimum value

Type
float

y2max
the third y axis maximum value

Type
float

y3min
the fourth y axis minimum value

Type
float

y3max
the fourth y axis maximum value

Type
float

hour_stop
the maximum x axis time value to plot

Type
float

ln0
the plotted PYPOWER bus voltage, color GREEN

Type
Line2D

ln1
the plotted EnergyPlus load, color RED

Type
Line2D

ln2lmp
the plotted PYPOWER locational marginal price (LMP), color BLUE

Type
Line2D

ln2auc
the plotted simple_auction cleared_price, color BLACK

Type
Line2D

ln3gld

the plotted sample-and-hold GridLAB-D substation load, color MAGENTA

Type
Line2D

ln3fncs

the plotted GridLAB-D substation load published via FNCS; may be zero if not published for the current animation frame, color CYAN

Type
Line2D

fig

animated Matplotlib figure hosted on the GUI

Type
Figure

ax

set of 4 xy axes to plot on

Type
Axes

canvas

a TCL Tk canvas that can host Matplotlib

Type
FigureCanvasTkAgg

bFNCSactive

True if a TESP simulation is running with other FNCS federates, False if not

Type
bool

bHELICSactive

True if a TESP simulation is running with other HELICS federates, False if not

Type
bool

OpenConfig()

Read the JSON configuration file for this monitor, and initialize the plot axes

Quit()

Shuts down this monitor, and also shuts down FNCS/HELICS if active

expand_limits(*v*, *vmin*, *vmax*)

Whenever a variable meets a vertical axis limit, expand the limits with 10% padding above and below

Parameters

- **v** (*float*) – the out of range value
- **vmin** (*float*) – the current minimum vertical axis value
- **vmax** (*float*) – the current maximum vertical axis value

Returns
the new vmin and vmax

Return type
float, float

kill_all()

Shut down all FNCS/HELICS federates in TESP, except for this monitor

launch_all()

Launches the simulators, initializes HELICS and starts the animated plots

launch_all_f()

Launches the simulators, initializes FNCS and starts the animated plots

on_closing()

Verify whether the user wants to stop TESP simulations before exiting the monitor

This monitor is itself a FNCS federate, so it can not be shut down without shutting down all other FNCS federates in the TESP simulation.

reset_plot()

update_plots(*i*)

This function is called by Matplotlib for each animation frame

Each time called, collect HELICS messages until the next time to plot has been reached. Then update the plot quantities and return the Line2D objects that have been updated for plotting. Check for new data outside the plotted vertical range, which triggers a full re-draw of the axes. On the last frame, finalize HELICS.

Args:
i (int): the animation frame number

update_plots_f(*i*)

This function is called by Matplotlib for each animation frame

Each time called, collect FNCS messages until the next time to plot has been reached. Then update the plot quantities and return the Line2D objects that have been updated for plotting. Check for new data outside the plotted vertical range, which triggers a full re-draw of the axes. On the last frame, finalize FNCS.

Args:
i (int): the animation frame number

`tesp_support.original.tesp_monitor.`**show_tesp_monitor**(*HELICS=True*)

Creates and displays the monitor GUI

tesp_support.original.tesp_monitor_ercot module

Presents a GUI to launch a TESP simulation and monitor its progress

This version differs from the one in *tesp_monitor*, in that the user can select the FNCS federate and topic to plot. The number of monitored plots is still fixed at 4.

Public Functions:

show_tesp_monitor
Initializes and runs the monitor GUI

References

Graphical User Interfaces with Tk

Matplotlib Animation

class tesp_support.original.tesp_monitor_ercot.**ChoosablePlot**(*master*, *color='red'*, *xLabel=''*, *topicDict={}*, ***kwargs*)

Bases: `Frame`

Hosts one Matplotlib animation with a selected variable to plot

Parameters

- **master** –
- **color** (*str*) – Matplotlib color of the line to plot
- **xLabel** (*str*) – label for the x axis
- **topicDict** (*dict*) – dictionary of FNCS topic choices to plot
- **kwargs** – arbitrary keyword arguments

topicDict

listOfTopics

root

the TCL Tk toolkit instance

Type
Tk

fig

animated Matplotlib figure hosted on the GUI

Type
Figure

ax

set of 4 xy axes to plot on

Type
Axes

xLabel

horizontal axis label

Type
str

yLabel

vertical axis label

Type
str

hrs

y

ymin

Type
float

ymax

Type
float

color

title

Type
str

ln

Type
Line2D

topicSelectionRow

topicLabel

combobox

voltageLabel

voltageBase

voltageBaseTextbox

onTopicSelected(*event*)

Change the GUI labels to match selected plot variable

Parameters
event –

class tesp_support.original.tesp_monitor_ercot.**TespMonitorGUI**(*master*)

Bases: object

Version of the monitor GUI that hosts 4 choosable plots

Parameters
master –

root

the TCL Tk toolkit instance

Type
Tk

top

the top-level TCL Tk Window

Type
Window

labelvar

used to display the monitor JSON configuration file path

Type
StringVar

plot0

first plot

Type
ChoosablePlot

plot1

second plot

Type
ChoosablePlot

plot2

third plot

Type
ChoosablePlot

plot3

fourth plot

Type
ChoosablePlot

topicDict

plots

Type
Frame

scrollbar

Type
Scrollbar

frameInCanvas

Type
Frame

canvas

a TCL Tk canvas that can host Matplotlib

Type
FigureCanvasTkAgg

bFNCSactive

True if a TESP simulation is running with other FNCS federates, False if not

Type
bool

CalculateValue(*topic*, *value*, *plot*)

Parses a value from FNCS to plot

Parameters

- **topic** (*str*) – the FNCS topic
- **value** (*str*) – the FNCS value
- **plot** (ChoosablePlot) – the plot that will be updated; contains the voltage base if needed

Returns
the parsed value

Return type
float

OpenConfig()

Read the JSON configuration file for this monitor, and initialize the plot axes

Quit()

Shuts down this monitor, and also shuts down FNCS if active

kill_all()

Shut down all FNCS federates in TESP, except for this monitor

launch_all()

Launches the simulators, initializes FNCS and starts the animated plots

onFrameConfigure(*event*)

Reset the scroll region to encompass the inner frame

on_closing()

Verify whether the user wants to stop TESP simulations before exiting the monitor

This monitor is itself a FNCS federate, so it can not be shut down without shutting down all other FNCS federates in the TESP simulation.

update_plots(*i*)

This function is called by Matplotlib for each animation frame

Each time called, collect FNCS messages until the next time to plot has been reached. Then update the plot quantities and return the Line2D objects that have been updated for plotting. Check for new data outside the plotted vertical range, which triggers a full re-draw of the axes. On the last frame, finalize FNCS.

Args:
i (int): the animation frame number

tesp_support.original.tesp_monitor_ercot.**show_tesp_monitor**()

Creates and displays the monitor GUI

tesp_support.original.tso_PYPOWER_f module

PYPOWER solutions under control of FNCS or HELICS for te30 and dsot, sgip1 examples

Public Functions:

pypower_loop
Initializes and runs the simulation.

tesp_support.original.tso_psst_f module

`tesp_support.original.tso_psst_f.`**`make_generator_plants`**(*ppc*, *renewables*)

tesp_support.sgip1 package

Transactive Energy Simulation Platform (TESP) Contains the python files for the SGIP1 analysis Plotting all SGIP1 variants on the same graph

Submodules

tesp_support.sgip1.compare_auction module

`tesp_support.sgip1.compare_auction.`**`MakePlotData`**(*root*)

`tesp_support.sgip1.compare_auction.`**`compare_auction`**()

tesp_support.sgip1.compare_csv module

`tesp_support.sgip1.compare_csv.`**`MakePlotData`**(*root*, *idx*)

`tesp_support.sgip1.compare_csv.`**`compare_csv`**(*idx*)

tesp_support.sgip1.compare_hvac module

`tesp_support.sgip1.compare_hvac.`**`MakePlotData`**(*root*)

`tesp_support.sgip1.compare_hvac.`**`compare_hvac`**()

tesp_support.sgip1.compare_prices module

`tesp_support.sgip1.compare_prices.`**`MakePlotData`**(*root*)

`tesp_support.sgip1.compare_prices.`**`UnitPrice`**(*p*, *c2*, *c1*, *c0*)

`tesp_support.sgip1.compare_prices.`**`compare_prices`**(*root*)

tesp_support.sgip1.compare_pypower module

tesp_support.sgip1.compare_pypower.**MakePlotData**(*root*)

tesp_support.sgip1.compare_pypower.**compare_pypower**()

tesp_support.valuation package

Example files for calculating and plotting valuation quantities in post-processing. Also see the Jupyter notebook examples.

Submodules

tesp_support.valuation.TransmissionMetricsProcessor module

class tesp_support.valuation.TransmissionMetricsProcessor.**TransmissionMetricsProcessor**(*case_name=''*, *case_path=''*)

Bases: object

bus_metrics = None

casename = ''

casepath = ''

gen_metrics = None

get_allBus_LMP()

get_allGen_CO2_emissions()

get_allGen_NOx_emissions()

get_allGen_SOx_emissions()

get_allGen_cost()

get_allGen_revenue()

get_bus_LMP(*bus_num*)

get_bus_metrics()

get_bus_metrics_at_bus(*bus_num*)

get_bus_metrics_at_time(*a_time_instance*)

get_bus_metrics_for_period(*start_time*, *end_time*)

get_cost_at_gen(*busNum*)

get_eachGen_CO2_emissions()

get_eachGen_NOx_emissions()

get_eachGen_SOx_emissions()

get_eachGen_cost()

get_eachGen_revenue()

get_gen_metrics()

get_gen_metrics_for_period(*start_time*, *end_time*)

loadAllMetricsFromJSONFiles(*case_name=''*, *case_path=''*)

loadBusMetricsFromNetCMFFile(*bus_metrics_netCMF*)

loadGenMetricsFromNetCMFFile(*gen_metrics_netCMF*)

saveBusMetricsToNetCMFFile(*new_bus_metrics_netCMF*)

saveGenMetricsToNetCMFFile(*new_gen_metrics_netCMF*)

tesp_support.weather package

TESP is the Transactive Energy Simulation Platform tesp_support for Weather

Submodules

tesp_support.weather.PSM_download module

Simple script to download PSM weather files and convert them to DAT files.

In the name of expediency, this script contains two functions that were copy and pasted from dsot_solar.py. I wish that those functions had been better abstracted when I wrote them but, it's not worth the time and effort to do so now. This script will likely ever only be run a few times as it is just creating input files and is not a part of the co-sim runtime.

The script does make use of PSMvstoDAT.py as a function, though. I had to make minor edits to that function as it was slightly non-abstracted and not completely stand-alone.

`tesp_support.weather.PSM_download.`**download_nsrdb_data**(*dso_meta*, *output_path*)

This function queries the NSRDB database over the web and pulls down and does a conversion.

Function pulls down the solar data down calls PSMv3toDAT to convert them to appropriate format.

Parameters

- **dso_meta** (*list*) – List of dicts with metadata associated with each DSO, specifically the site information
- **output_path** (*str*) –

`tesp_support.weather.PSM_download.`**parse_DSO_location**(*dso_metadata_path*, *worksheet_name*)

This function parses the DSO metadata which is contained in an Excel spreadsheet

Parameters

- **dso_metadata_path** (*str*) – Path to the Excel file containing the metadata to be parsed.
- **worksheet_name** (*str*) – Name of the worksheet in the Excel file containing the metadata

Returns

One dictionary per DSO with appropriate metadata captured.

Return type

list<dict>

tesp_support.weather.PSMv3toDAT module

This code reads in PSM v3 csv files obtained from https://maps.nrel.gov/nsrdb-viewer/.

The csv file includes 6 columns for the following, in addition to date and time information: temperature, humidity, DNI, DHI, pressure and wind_speed

`tesp_support.weather.PSMv3toDAT.weatherdat(psmv3csvfile, bus_str, location_str)`

Takes a weather csv file name obtained from NREL PSM v3 and does a conversion.

The function reads the file, converts the data to the desired units, and outputs dat file with the desired format

Parameters

- **psmv3csvfile** (*str*) – name of the file to be converted, without ext '.csv'
- **bus_str** (*str*) – id of the bus
- **location_str** (*str*) –

tesp_support.weather.TMY3toCSV module

Convert typical meteorological year version 3 (TMY3) data to comma separated values (CSV)

For common use by TESP agents.

Public Functions:

readtmy3
Read a TMY3 file in to a pandas dataframe, using python tkinter.

weathercsv
Converts TMY3 weather data to CSV in the requested time range.

weathercsv_cloudy_day
Converts weather data to cloudy days CSV in the requested time range.

`tesp_support.weather.TMY3toCSV.readtmy3(filename=None, coerce_year=None, recolumn=True)`

Read a TMY3 file in to a pandas dataframe.

Parameters

- **filename** (*object*) –
- **coerce_year** –
- **recolumn** –

Return type
dict, dict

`tesp_support.weather.TMY3toCSV.weathercsv(tmyfile, output_file, start_time, end_time, year)`

Converts TMY3 weather data to CSV in the requested time range

Parameters

- **tmyfile** (*str*) – the input TMY3 data file
- **output_file** (*str*) – the output CSV data file
- **start_time** (*str*) – the starting date and time of interest, like *2013-07-01 00:00:00*
- **end_time** (*str*) – the ending date and time of interest, like *2013-07-08 00:00:00*

- **year** (*int*) – the year of interest

`tesp_support.weather.TMY3toCSV.weathercsv_cloudy_day(start_time, end_time, output_file)`

Converts weather data to cloudy days CSV in the requested time range

Parameters

- **start_time** (*str*) – the starting date and time of interest, like *2013-07-01 00:00:00*
- **end_time** (*str*) – the ending date and time of interest, like *2013-07-08 00:00:00*
- **output_file** (*str*) – the output CSV data file

tesp_support.weather.TMYtoEPW module

Functions to convert Typical Meteorological Year data for EnergyPlus

The input must be in TMY2 format. If only TMY3 files are available, see the TMY3toTMY2_ansi.c program distributed with TESP under the [weather_path]/TMY2EPW/source_code directory.

Public Functions:

convert_tmy2_to_epw
Converts TMY2 file to EPW.

convert_tmy3_to_epw
Converts TMY3 file to EPW.

`tesp_support.weather.TMYtoEPW.convert_tmy2_to_epw(in_file_root, out_file_root=None)`

Converts TMY2 to EPW

Reads in_file_root.tmy2 and writes in_file_root.epw

Parameters

- **in_file_root** (*str*) – The input path and base file name, without extension
- **out_file_root** (*str*) – The output path and base file name, without extension

`tesp_support.weather.TMYtoEPW.convert_tmy3_to_epw(in_file_root, out_file_root=None)`

Converts TMY3 to EPW

Reads in_file_root.tmy3 and writes out_file_root.epw

Parameters

- **in_file_root** (*str*) – The input path and base file name, without extension
- **out_file_root** (*str*) – The output path and base file name, without extension

`tesp_support.weather.TMYtoEPW.removeZero(x)`

Helper function to strip leading '0' from a string

tesp_support.weather.weather_agent module

Weather Agent

This weather agent needs an WEATHER_CONFIG environment variable to be set, which is a json file.

`tesp_support.weather.weather_agent.convertTimeToSeconds(time)`

Convert time string with unit to integer in seconds

Parses the unit in day, hour, minute and second. It will not recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
represent the input time in second

Return type
int

`tesp_support.weather.weather_agent.deltaTimeToResmapleFreq(time)`

Convert time unit to a resampling frequency that can be recognized by pandas.DataFrame.resample()

Parses unit in day, hour, minute and second. It won't recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
time with resample frequency

Return type
str

`tesp_support.weather.weather_agent.findDeltaTimeMultiplier(time)`

Find the multiplier to convert delta_time to seconds

Parses unit in day, hour, minute and second. It won't recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
the multiplier to convert delta_time to seconds

Return type
int

`tesp_support.weather.weather_agent.startWeatherAgent(file)`

The weather agent publishes weather data as configured by the json file

Parameters
file (*str*) – the weather data file

`tesp_support.weather.weather_agent.usage()`

class `tesp_support.weather.weather_agent.weather_forecast(variable, period, W_dict)`

Bases: `object`

This object includes the error to a weather variable

Parameters

- **variable** (*str*) – Type of weather variable being forecasted
- **period** (*int*) – period of the sinusoidal bias
- **W_dict** (*dict*) – dictionary for specifying the generation of the error envelope

weather_variable

Type of weather variable being forecasted

Type
str

 # Type of error insertion

distribution

type of distribution –> 0 uniform;1 triangular;2 truncated normal the standard deviation is computed for 95% of values to be within bounds in a conventional normal distribution

Type
int

P_e_bias

pu maximum bias at first hour –> [0 to 1]

Type
float

P_e_envelope

pu maximum error from mean values –> [0 to 1]

Type
float

Lower_e_bound

pu of the maximum error at the first hour –> [0 to 1]

Type
float

 # Bias variable

biasM

sinusoidal bias for altering the error envelope

Type
float) (1 X period

Period_bias

period of the sinusoidal bias

Type
int

get_truncated_normal(*EL*, *EH*)

Truncated normal distribution

make_forecast(*weather*, *t=0*)

Include error to a known weather variable

Parameters

- **weather** (*float) (1 x desired number of hours ahead*) – known weather variable
- **t** (*int*) – time in hours

Returns
weather variable with included error ENV_U (float) (1 x desired number of hours ahead): envelope with bias upper bound ENV_l (float) (1 x desired number of hours ahead): envelope with bias lower bound

Return type
weather_f (float) (1 x desired number of hours ahead)

tesp_support.weather.weather_agent_f module

Weather Agent

This weather agent needs an WEATHER_CONFIG environment variable to be set, which is a json file.

`tesp_support.weather.weather_agent_f.convertTimeToSeconds(time)`

Convert time string with unit to integer in seconds

Parses the unit in day, hour, minute and second. It will not recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
represent the input time in second

Return type
int

`tesp_support.weather.weather_agent_f.deltaTimeToResmapleFreq(time)`

Convert time unit to a resampling frequency that can be recognized by pandas.DataFrame.resample()

Parses unit in day, hour, minute and second. It won't recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
time with resample frequency

Return type
str

`tesp_support.weather.weather_agent_f.findDeltaTimeMultiplier(time)`

Find the multiplier to convert delta_time to seconds

Parses unit in day, hour, minute and second. It won't recognize week, month, year, millisecond, microsecond or nanosecond, they can be added if needed.

Parameters
time (*str*) – time with unit

Returns
the multiplier to convert delta_time to seconds

Return type
int

`tesp_support.weather.weather_agent_f.startWeatherAgent(file)`

The weather agent publishes weather data as configured by the json file

Parameters
file (*str*) – the weather data file

`tesp_support.weather.weather_agent_f.usage()`

class `tesp_support.weather.weather_agent_f.weather_forecast(variable, period, W_dict)`

Bases: `object`

This object includes the error to a weather variable

Parameters

- **variable** (*str*) – Type of weather variable being forecasted
- **period** (*int*) – period of the sinusoidal bias
- **W_dict** (*dict*) – dictionary for specifying the generation of the error envelope

weather_variable

Type of weather variable being forecasted

Type
str

 # Type of error insertion

distribution

type of distribution –> 0 uniform;1 triangular;2 truncated normal the standard deviation is computed for 95% of values to be within bounds in a conventional normal distribution

Type
int

P_e_bias

pu maximum bias at first hour –> [0 to 1]

Type
float

P_e_envelope

pu maximum error from mean values –> [0 to 1]

Type
float

Lower_e_bound

pu of the maximum error at the first hour –> [0 to 1]

Type
float

 # Bias variable

biasM

sinusoidal bias for altering the error envelope

Type
float) (1 X period

Period_bias

period of the sinusoidal bias

Type
int

get_truncated_normal(*EL*, *EH*)

Truncated normal distribution

make_forecast(*weather*, *t=0*)

Include error to a known weather variable

Parameters

- **weather** (*float) (1 x desired number of hours ahead*) – known weather variable
- **t** (*int*) – time in hours

Returns
weather variable with included error ENV_U (float) (1 x desired number of hours ahead): envelope with bias upper bound ENV_l (float) (1 x desired number of hours ahead): envelope with bias lower bound

Return type
weather_f (float) (1 x desired number of hours ahead)

5.4 TESP Design Philosophy and Development Plans

5.4.1 Introduction

The Transactive Energy Simulation Platform (TESP) is a signature component of the Transactive Systems Program (TSP) funded by Chris Irwin in the Office of Electricity (OE) at the US Department of Energy (DOE). This simulation platform is an on-ramp into transactive energy simulations by providing an easy-to-install collection of simulators that have been pre-configured to exchange certain information via HELICS. In addition to the simulators several transactive agents have been developed by PNNL and included in TESP to allow for the simulation of, for example, residential HVAC systems participating in a transactive control scheme using an integrated wholesale-retail real-time energy market. In addition to the software proper, documention of the software is provided via a Read The Docs website. TESP has served as the foundation for a few studies studies including PNNL's Distribution System Operator + Transmission (DSO+T), and a transactive energy blockchain demonstration.

The key goal of TESP has been to provide a means of reducing the barrier to entry for evaluating transactive systems through time-series simulation. Transactive energy (TE), by its nature, tries to look more holistically at how systems operate (across traditional analysis and jurisdictional divides) and as a result tends to require information exchange across simulator domains. This creates several challenges for those new to TE.

- Knowledge of multiple analysis tools to replicate the behavior of the specific subsystems being connected is required
- Knowledge of how to connect multiple tools on the co-simulation platform of choice is required

For those with a TE mechanism they would like to evaluate in-hand, the barrier of learning and integrating new analysis tools can easily be overwhelming and result in the researcher in question either abandoning the idea or evaluating their mechanism by another method which may or may not be as comprehensive as would be possible with an integrated simulation and analysis environment.

In an ideal world, TESP would be able to directly and comprehensively address these challenges. It provides an integrated transmission, generation, and distribution power system simulation capability with sufficient detail to allow for simulation of classical transactive mechanisms (such as HVAC participation in an integrated wholesale-retail transactive real-time energy market). Using one of the built-in examples, it is easy to imagine a researcher with a similar transactive idea or wanting to evaluate the provided transactive system in a slightly different environment (say, one with high penetration of electric vehicles) being able to make relatively simple modifications to the provided models and scripts and perform the necessary analysis.

That's the dream: take all the simulation management complexity away from TE researchers and allow them to focus on the details of their unique ideas, evaluating them quickly and easily in TESP.

5.4.2 Challenges of a Generic TESP

Achieving the dream of a generic TESP where any transactive mechanism can be easily evaluated is not trivial.

TE can be Arbitrarily Specific

The world of TE is very large. At its core, it is a merger of economics and control theory and draws on the rich foundation from both. Development of TE mechanisms have the blessing of being able to utilize the theory of both of these domains to develop mechanisms that better capture value, better account for uncertainties, and are more easily implemented by practitioners. From a simulation environment, this blessing becomes a curse: any given transactive mechanism may require modeling new features and/or facilitating new information exchange between simulators to achieve a meaningful simulation test environment for said mechanism.

What is required to support a given transactive mechanism is arbitrarily complex and may or may not easily align with the existing software suite and its underlying architecture. TESP cannot both be both usable as a plug-and-play TE evaluation environment where users can expect to avoid the complexity of the underlying software and just use it out-of-the-box while simultaneously supporting arbitrarily general TE mechanisms.

Modifying TESP can Quickly Become Non-Trivial

Since TESP will never be able to provide the no-effort on-ramp to any generic TE mechanism, what can it do to allow users to customize it more easily? To some extent, this question is fighting against the goals of TESP, providing a no-hassle method of evaluating new TE mechanisms. Every line of code and configuration that a user has to modify is one more than we would ideally like. Given the reality of what TESP would ideally support, though, expecting a user to have no hand in the coding pie is unreasonable. So what challenges would a TESP user face in trying to customize TESP?

The software toolset that TESP traditionally uses to implement TE mechanisms is specific. We use PyPower to model and solve transmission system power flows (and perform economic dispatch, if necessary), AMES/PSST to perform wholesale market operations, GridLAB-D to model distribution systems, solve their powerflows, and model residential customers and limited number of commercial customers, Energy+ to model a broader set of commercial customers (though not as well as we'd like), custom Python TE agents to perform the transactive control, HELICS to provide the integration between all these simulators and Python scripts to build the simulation cases of interest. Depending on the transactive mechanism, a TESP user may require modification in all of these tools and scripts (adding the DSO+T DA energy functionality did).

Analysis Needs Vary

Under the best case scenario, a TESP user may desire to perform an analysis that TESP was specifically designed to support. It could easily be, though, that the user has different analysis needs than that originally imagined when the TESP example was built. Take, for example, a user that wants to evaluate the impacts of rooftop solar PV on the efficiency of real-time transactive energy markets, *the* classic transactive mechanism. It could be that this analyst wants to evaluate hundreds or even thousands of specific scenarios and is willing to given up some degree of modeling fidelity to crank through the simulations faster. Ideally TESP would be able to take an existing example with high fidelity and provide a way to quickly simplify it to speed up the simulation time.

TESP, though, does not support aribtrary levels of modeling fidelity. Even if the TE mechanism is well supported, to perform specific analysis significantly different models and simulation case constructions scripts may be required. This, again, requires the users to get their hands dirty to customize TESP to suit their analysis needs and likely requires a greater-than-cursory understanding of the software suite.

5.4.3 Addressing Challenges: The Design Philosophy of TESP

Given these challenges, the only way for TESP to be successful is to carefully define "success" in a way that makes the scope and scale of the software's goals clear. To that end, a few guiding principles have been defined to articulate the design philosophy of TESP.

TESP Targets Expert Users

Given the generic nature of how TESP is expected to be used and the level of expertise in designing and implementing transactive systems, TESP expects a high degree of skill from its users. Ideally users (or teams of users) would have sufficient experience with simulation tools deployed in TESP that constructing an configuring an appropriate set of tools for a given analysis would be a tractable task. To learn to use TESP well will likely require a commitment on the part of a new user that is measured in weeks or months rather than days.

Though this is not desirable, there are many other complex software analysis packages (*e.g.* DOE's NEMS, Seimen's PSS/E, Energy Exemplar's PLEXOS) that require similar or greater levels of time and effort investment to learn to use well. For analysis that are very similar to those distributed with TESP, the barrier to entry is much lower as much of the analysis design and construction is baked into that example.

TESP Requires Excellent Documentation

Because of the complexity of the simulation and analysis toolset, if TESP is to have any hope of being used (even by those that are implementing it), the documentation of the capabilities, the examples, demonstrations, and models that are distributed with it, and the code itself all need to be first-class. If we expect users to use TESP knowing that they will have to modify existing code and craft new ones to achieve new analysis goals, we need to enable them to do so. How the TESP API is used, what assumptions went into certain models, how the simulators were tied together in co-simulation for a certain example, all of this needs to be clearly spelled out. Expecting users to become proficient through reading source code is not realistic if we expect the adoption of TESP to grow.

Built-In Examples Show How Things Could Be Done

To help bridge the gap between an analysis goal and the user's unfamiliarity with the TESP toolset, a broad suite of capability demonstrations and example analysis need to be included with TESP. Capability demonstrations show how to user specific simulation tools, APIs, or models so that new users can understand the capabilities of TESP and how to use specific features. Example analysis are implementations in TESP of analysis that have been done with TESP. Often these analysis will have publications and a simplified version of the analysis will be implemented in TESP. The examples differ from the demonstrations in that they are analysis that are intended to produce meaningful results whereas the demonstrations are more about showing how to use a specific part of TESP. Being legitimate analysis, the examples will generally have a broader set of documentation that shows how the analysis as a whole was conducted and not just, say, how the co-simulation was configured.

Building Common Functions into the TESP API

TESP provides the greatest value to users when it provides easier ways of doing hard things. Some of this comes through the built-in tool integration with HELICS but much of it is realized through the careful definition of the TESP API. Though the possible transactive systems analysis space is very large, there are common functions or standard practices that can be implemented through the TESP API that will standardize TESP in ways that will increase understanding and portability across specific analysis. Examples of such common functions could be data collection, post-processing to produce common metrics, standardize models and scripts to manipulate them. Some of these API calls may be specific to some common analysis.

5.4.4 The Path Forward for TESP

The foundation of TESP has been laid in that there are a working set of simulation tools that have been tied together in a co-simulation platform and a few full-scale, publishable analysis have been conducted using it. This said, for TESP to realize its potential as an evaluation platform of transactive systems significant development is still required will take many years at the current funding rate. Below are some of the major development efforts planned for TESP in the coming years:

TESP API

As was previously discussed, despite the general TE system analysis TESP desires to support, there are general functionalities that would allow TESP users to more efficiently build the simulation and conduct the analysis. Defining the contents of this API and developing it are a significant undertaking as will be the conversion of the existing TESP codebase to use the new API. The definition of the new API is expected to begin in FY22. Some preliminary thinking has identified the following areas as candidates for inclusion in the TESP API:

- Standardized, database-oriented data collection for time series inputs to and outputs from the co-simulation. helics_cli already uses a sqlite database that for data collection and this database could be co-opted for TESP as a whole.
- Definition of a standardized GridLAB-D model structure that would allow feeder_generator.py to easily add supported object values as well as edit models that have already contain some of these objects
- Definition and implementation of common metrics calculated from standard data collected from the co-simulation. Examples include total energy consumption, total energy cost, total economic surplus, bulk power system generator revenue, and indoor comfort
- Standard DER device agents that provide estimated energy consumption for arbitrary periods into the future. This information can be used by a HEMS to interact with the TE system to acquire the appropriate energy (or curtail the DER operation)
- Forecasters for common signals used by DER agents to create their forecasted device loads

- Standardized double-auction implementation

Reference Implementations of Common TE Systems

Where TESP capability demonstrations are intended to be small in scale and show off specific TESP features and the example analysis are versions of actual studies done using TESP (and likely not maintained with API updates), there could be space for a middle-ground: reference implementations of common TE systems. These would be maintained as TESP evolves and would serve as starting points for users who want to do more specific studies. Candidates for reference implementations include integrated wholesale-retail real-time energy markets, and integrated wholesale-retail real-time and day-ahead energy markets. These reference implementations could also include reference implementations of specific components such as a HEMS, a double-auction market operator, prepare_case scripts that build the co-simulation and perform the analysis.

Full adoption of HELICS v3

HELICS v3 provides a wide variety of features and tools that would simplify many of the co-simulation mechanics in TESP. Using helics_cli to launch our co-simulations (instead of more complicated shell scripts), using message handles where appropriate rather than just value handles to support easy inclusion of communication system effects, and possibly using the helics_cli database for general TESP data collection.

Improved Commercial and Industrial Models

The residential modeling provided by GridLAB-D has been used for many years of transactive studies and shown to be relatively effective. There's always ways in which the models and typical parameter values can be improved and updated but generally speaking, the modeling is good enough. The commercial building modeling has been difficult as the go-to simulation tool, EnergyPlus, does not handle the fast (minute-scale) dynamics that transactive systems typically operate under. GridLAB-D has a very limited set of commercial structures it models but these do not represent the diversity seen in the real-world. And the industrial load modeling is non-existent and much more complex. There is a lot of work to be done if TESP is going to reflect the broad load landscape.

Communication System Modeling

Transactive systems (and the smart grid in general) rely on communication systems to operate but the modeling of these communication systems is challenging. A generic transactive system protocol stack does not exist and the most common modeling tools (*e.g.* ns-3, Omnet++) are more appropriate for protocol development than full system performance modeling (the models are generally much more detailed than required for transactive system analysis). A clear understanding of the goal of including a communication system model in a transactive system analysis needs to be articulated and appropriate protocols and simulation tools need to be identified or developed.

Standardize Capacity Expansion Modeling

Often, the biggest cost savings transactive energy provides is in capital cost savings, that is, in the power plants and transmission lines that don't need to get built (or are built much later) because transactive energy has been able to effectively manage the load. For analysis that want to capture these effects in simulation, a means of evolving the bulk power system under a given management philosophy (business as usual, time-of-use tarifs, transactive real-time tarifs) is an important part of creating a comprehensive apples-to-apples comparison. There are no tools in TESP to allow the run-time evolution of the power system and no direct integration with any of the popular capacity expansion tools to allow their outputs to easily fit into the TESP bulk power system modeling. This capability is an important missing piece for high-quality TE system evaluations.

5.5 Bibliography

5.6 TESP License

CHAPTER

SIX

ARCHIVES

Archived build and install instructions

6.1 Building on Ubuntu

This procedure builds all components from scratch. If you've already built GridLAB-D on your machine, please take note of the specific GitHub branch requirements for TESP:

- develop for GridLAB-D
- feature/opendss for FNCS
- fncs_9.3.0 for EnergyPlus
- master for HELICS 2.0

You may also need to upgrade the gcc and g++ compilers. This build procedure has been tested with Ubuntu 18.04 LTS and gcc/g++ 7.3.0.

When you finish the build, try *TESP Demonstrations and Examples*.

6.1.1 Preparation - Virtual Machine or Windows Subsystem for Linux (WSL)

Linux can be installed on Windows (or Mac) using a virtual machine (VM), such as VirtualBox from https://www.virtualbox.org/ The following instructions have been written for the example of installing Ubuntu 18.04 under VirtualBox for Windows. The same instructions also work for Ubuntu 18.04 under VMWare Fusion for Mac OS X.

Another option is to use the WSL feature built in to Windows, as described at https://github.com/michaeltreat/Windows-Subsystem-For-Linux-Setup-Guide. WSL does not support graphical applications, including the Eclipse IDE, but it may have other advantages for building and running command-line tools, like TESP and GridLAB-D. The following instructions work for Ubuntu 18.04 set up that way under WSL, with a few suggested changes to the TESP build:

- from the *cdwr* prompt, use *mkdir usrc* instead of *mkdir ~/src* before checking out repositories from GitHub. This makes it easier to keep track of separate source trees for Windows and Linux, if you are building from both on the same machine.
- the first step of *sudo apt-get install git* is not necessary
- when building the Java 10 binding for FNCS, you have to manually copy the fncs.jar and libFNCSjni.so to the correct place. The paths are different because of how WSL integrates the Windows and Linux file systems
- for HELICS bindings, add JAVAPATH to *~/.profile* instead of *~/.bashrc*

TESP and its component simulators do not perform as well under WSL1 as they do inside a VM. Performance testing has not been done with WSL2.

6.1.2 Preparation - Build Tools and Java

```
# build tools
sudo apt-get -y install git-lfs
sudo apt-get -y install build-essential
sudo apt-get -y install autoconf
sudo apt-get -y install libtool
sudo apt-get -y install libjsoncpp-dev
sudo apt-get -y install gfortran
sudo apt-get -y install cmake
sudo apt-get -y install subversion
# Java support
sudo apt-get -y install openjdk-11-jre-headless
sudo apt-get -y install openjdk-11-jdk-headless
sudo ln -s /usr/lib/jvm/java-11-openjdk-amd64 /usr/lib/jvm/default-java
# for HELICS and FNCS
sudo apt-get -y install libzmq5-dev
sudo apt-get -y install libczmq-dev
# for GridLAB-D
sudo apt-get -y install libxerces-c-dev
sudo apt-get -y install libsuitesparse-dev
# end users replace libsuitesparse-dev with libklu1, which is licensed LGPL
# for AMES market simulator
sudo apt-get -y install coinor-cbc
# if not using miniconda (avoid Python 3.7 on Ubuntu for now)
sudo apt-get -y install python3-pip
sudo apt-get -y install python3-tk
```

6.1.3 Preparation - Python 3 and Packages

If you didn't previously install Python 3 using apt, which is the recommended method and version for TESP, please download and execute a Linux install script for Miniconda (Python*3.7*+) from https://docs.conda.io/en/latest/miniconda.html The script should not be run as root, otherwise, you won't have permission to install Python packages. After the script configures Conda please re-open the Ubuntu terminal as instructed.

With Python 3 available, install and test the TESP packages:

```
pip3 install tesp_support --upgrade
opf
```

In order to install psst:

```
pip3 install psst --upgrade
```

6.1.4 Checkout PNNL repositories from github

As noted above, we suggest *mkdir usrc* instead of *mkdir ~/src* on WSL.

```
mkdir ~/src
cd ~/src
git config --global user.name "your user name"
git config --global user.email "your email"
git config --global credential.helper store
git clone -b feature/opendss https://github.com/FNCS/fncs.git
git clone -b master https://github.com/GMLC-TDC/HELICS-src
git clone -b develop https://github.com/gridlab-d/gridlab-d.git
git clone -b fncs_9.3.0 https://github.com/FNCS/EnergyPlus.git
git clone -b develop https://github.com/pnnl/tesp.git
git clone https://gitlab.com/nsnam/ns-3-dev.git
git clone https://github.com/ames-market/psst.git
svn export https://github.com/gridlab-d/tools/branches/klu-build-update/solver_klu/
↪source/KLU_DLL
```

6.1.5 Choosing and Configuring the Install Directories

You must define the environment variable *$TESP_INSTALL*, which will receive the TESP build products, examples and common data files. */opt/tesp* is suggested.

It's possible, but not recommended, to set *$TESP_INSTALL* as /usr/local. There are a few reasons not to:

1. It would result in shared TESP data files and examples being copied to /usr/local/share
2. It complicates building the Linux installer and Docker images
3. The simulators install properly to /usr/local by default, but you still have to explicity set $TESP_INSTALL for the example scripts to run properly.

The following examples are for Ubuntu; other flavors of Linux may differ.

For Ubuntu in a *virtual machine*, first edit or replace the */etc/environment* file. This is not a script file, and it doesn't support the $variable replacement syntax. If using $TESP_INSTALL, it has to be spelled out on each line, e.g.:

```
TESP_INSTALL="/opt/tesp"
PYHELICS_INSTALL="/opt/tesp"
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/opt/tesp/bin:/opt/
↪tesp:/opt/tesp/PreProcess:/opt/tesp/PostProcess"
GLPATH="/opt/tesp/lib/gridlabd:/opt/tesp/share/gridlabd"
CXXFLAGS="-I/opt/tesp/share/gridlabd"
JAVAPATH="/opt/tesp/java"
```

Log out and log back in to Ubuntu for these */etc/environment* changes to take effect.

For Ubuntu in *WSL*, all changes are made to *~/.profile*.

```
export TESP_INSTALL="/opt/tesp"
export PATH="$PATH:$TESP_INSTALL:$TESP_INSTALL/bin:$TESP_INSTALL/PreProcess:$TESP_
↪INSTALL/PostProcess"
export GLPATH="$TESP_INSTALL/lib/gridlabd:$TESP_INSTALL/share/gridlabd"
export CXXFLAGS="-I$TESP_INSTALL/share/gridlabd"
export JAVAPATH="$TESP_INSTALL/java:$JAVAPATH"
```

Afterward, close and reopen the Ubuntu terminal for these changes to take effect.

The environment variable, CXXFLAGS, does not conflict with CXXFLAGS passed to various build tools. Only GridLAB-D uses the CXXFLAGS environment variable, and you should not use the variable append mechanism, i.e., :$CXXFLAGS, with it. This variable enables all of the GridLAB-D autotest cases to pass.

6.1.6 FNCS and HELICS

To build the shared libraries for FNCS with Python bindings:

```
cd ~/src/fncs
autoreconf -if
./configure 'CXXFLAGS=-w -O2' 'CFLAGS=-w -O2' --prefix=$TESP_INSTALL
# leave off --prefix if using the default /usr/local
make
sudo make install
```

To build the Java interface for version 10 or later, which has *javah* replaced by *javac -h*:

```
cd java
make
sudo -E make install
```

To build HELICS with Java bindings:

```
cd ~/src/HELICS-src
mkdir build
cd build
cmake -DBUILD_JAVA_INTERFACE=ON -DBUILD_SHARED_LIBS=ON \
      -DJAVA_AWT_INCLUDE_PATH=NotNeeded -DHELICS_DISABLE_BOOST=ON \
      -DCMAKE_INSTALL_PREFIX=$TESP_INSTALL -DCMAKE_BUILD_TYPE=Release ..
# leave off -DCMAKE_INSTALL_PREFIX if using the default /usr/local
git submodule update --init
make -j4
sudo make install
```

Test that HELICS and FNCS start:

```
sudo ldconfig
helics_player --version
helics_recorder --version
fncs_broker --version # look for the program to start, then exit with error
```

Install HELICS Python 3 bindings for a version that exactly matches the local build:

```
pip3 install helics==2.6.1
# where 2.6.1 came from helics_player --version
```

Then test HELICS from Python 3:

```
python3
>>> import helics
>>> helics.helicsGetVersion()
>>> quit()
```

6.1.7 GridLAB-D

To build the KLU solver:

```
cd ~/src/KLU_DLL
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$TESP_INSTALL ..
# replace $TESP_INSTALL with /usr/local if using the default
sudo -E env "PATH=$PATH" cmake --build . --target install
```

To link with both FNCS and HELICS, and run the autotest suite:

```
cd ~/src/gridlab-d
autoreconf -isf

# in the following, --with-fncs and --with-helics can not be left blank, so use either
↪$TESP_INSTALL or /usr/local for both
# leave off --prefix if using the default /usr/local
./configure --prefix=$TESP_INSTALL --with-fncs=$TESP_INSTALL --with-helics=$TESP_INSTALL␣
↪--enable-silent-rules 'CFLAGS=-w -O2' 'CXXFLAGS=-w -O2 -std=c++14' 'LDFLAGS=-w'
# for debugging use 'CXXFLAGS=-w -g -O0' and 'CFLAGS=-w -std=c++14 -g -O0' and 'LDFLAGS=-
↪w -g -O0'

make
sudo make install
gridlabd --validate
```

6.1.8 EnergyPlus

These following instructions install EnergyPlus with FNCS linkage and key portions of the retail v9.3 installation.

```
cd ~/src/EnergyPlus
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$TESP_INSTALL -DBUILD_FORTRAN=ON -DBUILD_PACKAGE=ON -
↪DENABLE_INSTALL_REMOTE=OFF ..
# leave off -DCMAKE_INSTALL_PREFIX if using the default /usr/local
make -j4
sudo make install
```

6.1.9 Build eplus_agent

```
cd ~/src/tesp/src/energyplus
# the following steps are also in go.sh
autoheader
aclocal
automake --add-missing
autoconf
./configure --prefix=$TESP_INSTALL --with-fncs=$TESP_INSTALL 'CXXFLAGS=-w -O2' 'CFLAGS=-
↪w -O2'
```

(continues on next page)

(continued from previous page)

```
# leave off --prefix and --with-fncs if using the default /usr/local
make
sudo make install
```

6.1.10 Build EnergyPlus Weather File Utility

```
cd ~/src/tesp/support/weather/TMY2EPW/source_code
sudo -E make
```

6.1.11 Build ns3 with HELICS

First, in order to build ns-3 with Python bindings, we need to install the Python binding generator that it uses from source, and keep it updated for ns-3. The version from *pip3 install pybindgen* is not kept up to date for ns-3 builds.

```
cd ~/src
git clone https://github.com/gjcarneiro/pybindgen.git
cd pybindgen
sudo python3 setup.py install
```

Then, we can build ns-3, install that into the same location as other parts of TESP, and test it:

```
cd ~/src/ns-3-dev
# first build: use the following command for HELICS interface to ns3:
git clone -b feature/13b https://github.com/GMLC-TDC/helics-ns3 contrib/helics
# subsequent builds: use the following 3 commands to update HELICS interface code:
# cd contrib/helics
# git pull
# cd ../..
# then configure, build and test ns3 with the HELICS interface
# --with-helics may not be left blank, so use either $TESP_INSTALL or /usr/local
./waf distclean
./waf configure --prefix=$TESP_INSTALL --with-helics=$TESP_INSTALL --build-
↪profile=optimized --disable-werror --enable-logs --enable-examples --enable-tests
./waf build
sudo ./waf install
./test.py
```

6.1.12 Prepare for Testing

This command ensures Ubuntu will find all the new libraries, before you try *TESP Demonstrations and Examples*.

```
# if using $TESP_INSTALL, edit the helper file tesp_ld.conf accordingly and then:
sudo cp ~src/tesp/install/Linux/helpers/tesp_ld.conf /etc/ld.so.conf.d
# then, regardless of whether the previous command was necessary:
sudo ldconfig
```

In case you have both Python 2 and Python 3 installed, the TESP example scripts and post-processing programs only invoke *python3*.

6.1.13 Building Documentation

In order to build the documentation for ReadTheDocs:

```
pip3 install recommonmark --upgrade
pip3 install sphinx-jsonschema --upgrade
pip3 install sphinx_rtd_theme --upgrade
# sphinxcontrib-bibtex 2.0.0 has introduced an incompatibility
pip3 install sphinxcontrib-bibtex==1.0.0
cd ~/src/tesp/doc
make html
```

Changes can be previewed in ~/src/tesp/doc/_build/html/index.rst before pushing them to GitHub. There is a trigger on ReadTheDocs that will automatically rebuild public-facing documentation after the source files on GitHub change.

6.1.14 Deployment - Ubuntu Installer

The general procedure will be:

1. Build TESP, installing to the default /opt/tesp
2. Clear the outputs from any earlier testing of the examples in your local repository
3. Deploy the shared files, which include examples, to /opt/tesp/share
4. Build opendsscmd to /opt/tesp/bin and liblinenoise.so to /opt/tesp/lib. (One source is the GridAPPS-D project repository under ~/src/CIMHub/distrib. Two copy commands are included in deploy.sh)
5. Make a sample user working directory, and auto-test the examples
6. Build and upload a Linux script installer using VMWare InstallBuilder. This is primarly based on the contents of /opt/tesp

Under ~/src/tesp/install/helpers, the following scripts may be helpful:

1. provision.sh; runs sudo apt-get for all packages needed for the build
2. gitclone.sh; clones all repositories need for the build
3. clean_outputs.sh; removes temporary output from the example directories
4. deploy.sh; copies redistributable files to /opt/tesp, invoking:
    1. deploy_ercot.sh; copies the ERCOT test system files to /opt/tesp
    2. deploy_examples.sh; copies the example files to /opt/tesp
    3. deploy_support.sh; copies the taxonomy feeder, reference building, sample weather, helper scripts and other support files to /opt/tesp
5. environment.sh; sets TESP_INSTALL and other environment variables
6. tesp_ld.conf; copy to /etc/ld.so.conf.d so Ubuntu fill find the shared libraries TESP installed
7. make_tesp_user_dir.sh; creates a working directory under the users home, and makes a copy of the shared examples and ERCOT test system.

6.1.15 Deployment - Docker Container

The Windows and Mac OS X platforms are supported now through the Docker container *tesp_core*. As pre-requisites for building this container:

1. Install Docker on the build machine, following https://docs.docker.com/engine/install/ubuntu/
2. Build and test the Ubuntu installer as described in the previous subsection. By default, InstallBuilder puts the installer into *~/src/tesp/install/tesp_core*, which is the right place for a Docker build.

This Docker build process layers two images. The first image contains the required system and Python packages for TESP, on top of Ubuntu 18.04, producing *tesp_foundation*. (In what follows, substitute your own DockerHub user name for *temcderm*)

```
cd ~/src/tesp/install/tesp_foundation
sudo docker build -t="temcderm/tesp_foundation:1.0.2" .
```

This process takes a while to complete. The second image starts from *tesp_foundation* and layers on the TESP components. Primarily, it runs the Linux installer script inside the Docker container. It will check for current versions of the packages just built into *tesp_foundation*, but these checks usually return quickly. The advantage of a two-step image building process is that most new TESP versions can start from the existing *tesp_foundation*. The only exception would be if some new TESP component introduces a new dependency.

```
cd ~/src/tesp/install/tesp_core
sudo docker build -t="temcderm/tesp_core:1.0.2" .
```

When complete, the layered image can be pushed up to Docker Hub.

```
cd ~/src/tesp/install/tesp_core
sudo docker push temcderm/tesp_core:1.0.2
```

6.1.16 DEPRECATED: MATPOWER, MATLAB Runtime (MCR) and wrapper

This procedure to support MATPOWER is no longer used in TESP at PNNL, but it may be useful to others working with TESP and MATPOWER.

```
cd ~/src/tesp/src/matpower/ubuntu
./get_mcr.sh
mkdir temp
mv *.zip temp
cd temp
unzip MCR_R2013a_glnxa64_installer.zip
./install  # choose /usr/local/MATLAB/MCR/v81 for installation target directory
cd ..
make

# so far, start_MATPOWER executable is built
# see MATLAB_MCR.conf for instructions to add MCR libraries to the Ubuntu search path
# unfortunately, this creates problems for other applications, and had to be un-done.
# need to investigate further:
# see http://sgpsproject.sourceforge.net/JavierVGomez/index.php/Solving_issues_with_
↪GLIBCXX_and_libstdc%2B%2B
```

6.2 Building on Mac OS X (DEPRECATED)

This procedure builds all components from scratch. It was last used in December 2019.

If you've already built GridLAB-D on your machine, please take note of the specific GitHub branch requirements for TESP:

- feature/1173 for GridLAB-D
- develop for FNCS
- fncs-v8.3.0 for EnergyPlus

The Mac OS X build procedure is very similar to that for Linux, and should be executed from the Terminal. For consistency among platforms, this procedure uses gcc rather than clang. It's also assumed that Homebrew has been installed.

It may also be necessary to disable system integrity protection (SIP), in order to modify contents under */usr*. Workarounds to set the *LD_LIBRARY_PATH* and *DYLD_LIBRARY_PATH* environment variables have not been tested successfully.

When you finish the build, try *TESP Demonstrations and Examples*.

6.2.1 Build GridLAB-D

Follow these directions:

```
http://gridlab-d.shoutwiki.com/wiki/Mac_OSX/Setup
```

6.2.2 Install Python Packages, Java, updated GCC

```
# install Python 3.7+ from Conda
# tesp_support, including verification of PYPOWER dependency
pip install tesp_support
opf

brew install gcc-9

# also need Java, Cmake, autoconf, libtool
```

6.2.3 Checkout PNNL repositories from github

```
mkdir ~/src
cd ~/src
git config --global (specify user.name, user.email, color.ui)
git clone -b develop https://github.com/FNCS/fncs.git
git clone -b feature/1173 https://github.com/gridlab-d/gridlab-d.git
git clone -b fncs-v8.3.0 https://github.com/FNCS/EnergyPlus.git
git clone -b develop https://github.com/pnnl/tesp.git
git clone -b master https://github.com/GMLC-TDC/HELICS-src
```

6.2.4 FNCS with Prerequisites (installed to /usr/local)

Your Java version may have removed *javah*. If that's the case, use *javac -h* instead.

```
brew install zeromq
brew install czmq

cd ../fncs
autoreconf -isf
./configure --with-zmq=/usr/local --with-czmq=/usr/local 'CPP=gcc-9 -E' 'CXXPP=g++-9 -E'
↪'CC=gcc-9' 'CXX=g++-9' 'CXXFLAGS=-w -O2 -mmacosx-version-min=10.12' 'CFLAGS=-w -O2 -
↪mmacosx-version-min=10.12'
make
sudo make install

cd java
mkdir build
cd build
cmake -DCMAKE_C_COMPILER="gcc-9" -DCMAKE_CXX_COMPILER="g++-9" ..
make
# copy jar and jni library to  tesp/examples/loadshed/java
```

6.2.5 HELICS (installed to /usr/local, build with gcc9)

To build HELICS:

```
cd ~/src/HELICS-src
rm -r build
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX="/usr/local" -DBUILD_PYTHON_INTERFACE=ON -DBUILD_JAVA_
↪INTERFACE=ON -DBUILD_SHARED_LIBS=ON -DJAVA_AWT_INCLUDE_PATH=NotNeeded -DHELICS_DISABLE_
↪BOOST=ON -DCMAKE_C_COMPILER=/usr/local/bin/gcc-9 -DCMAKE_CXX_COMPILER=/usr/local/bin/
↪g++-9 ../
make clean
make -j 4
sudo make install
```

To test HELICS:

```
helics_player --version
helics_recorder --version
ipython
import helics
helics.helicsGetVersion()
quit
```

Add this to .bash_profile

```
export PYTHONPATH=/usr/local/python:$PYTHONPATH
```

6.2.6 GridLAB-D with Prerequisites (installed to /usr/local)

If you encounter build errors with GridLAB-D, please try adding *-std=c++14* to *CXXFLAGS*.

```
brew install xerces-c

cd ~/src/gridlab-d
autoreconf -isf

./configure --with-fncs=/usr/local --with-helics=/usr/local --enable-silent-rules
↪'CPP=gcc-9 -E' 'CXXPP=g++-9 -E' 'CC=gcc-9' 'CXX=g++-9' 'CXXFLAGS=-O2 -w -std=c++14'
↪'CFLAGS=-O2 -w' LDFLAGS='-w'

sudo make
sudo make install
# TODO - set the GLPATH?
gridlabd --validate
```

6.2.7 ns-3 with HELICS

```
# consider -g flags on CXX, C and LD if debugging
cd ~/src
git clone https://gitlab.com/nsnam/ns-3-dev.git
cd ns-3-dev
git clone https://github.com/GMLC-TDC/helics-ns3 contrib/helics
./waf configure --with-helics=/usr/local --disable-werror --enable-examples --enable-
↪tests 'CPP=gcc-9 -E' 'CXXPP=g++-9 -E' 'CC=gcc-9' 'CXX=g++-9' 'CXXFLAGS=-w -std=c++14'
↪'CFLAGS=-w' LDFLAGS='-w'
./waf build
```

6.2.8 EnergyPlus with Prerequisites (installed to /usr/local)

```
cd ~/src/EnergyPlus
mkdir build
cd build
cmake -DCMAKE_C_COMPILER="gcc-9" -DCMAKE_CXX_COMPILER="g++-9" ..
make

# Before installing, we need components of the public version, including but not limited
#   to the critical Energy+.idd file
# The compatible public version is at https://github.com/NREL/EnergyPlus/releases/tag/v8.
↪3.0
# That public version should be installed to /usr/local/EnergyPlus-8-3-0 before going␣
↪further

sudo make install

# Similar to the experience with Linux and Windows, this installation step wrongly puts
#  the build products in /usr/local instead of /usr/local/bin and /usr/local/lib
#  the following commands will copy FNCS-compatible EnergyPlus over the public version
```

(continues on next page)

(continued from previous page)

```
cd /usr/local
cp energyplus-8.3.0 bin
cp libenergyplusapi.8.3.0.dylib lib

# if ReadVarsESO not found at the end of a simulation, try this
/usr/local/EnergyPlus-8-3-0$ sudo ln -s PostProcess/ReadVarsESO ReadVarsESO
```

6.2.9 Build eplus_agent

```
cd ~/src/tesp/src/energyplus
# the following steps are also in go.sh
autoheader
aclocal
automake --add-missing
# edit configure.ac to use g++-9 on Mac
autoconf
./configure --prefix=/usr/local --with-zmq=/usr/local --with-czmq=/usr/local
make
sudo make install
```

6.3 Building on Windows (DEPRECATED)

This procedure builds all components from scratch. It was last used in December 2019.

If you've already built GridLAB-D on your machine, please take note of the specific GitHub branch requirements for TESP:

- feature/1173 for GridLAB-D
- develop for FNCS
- fncs-v8.3.0 for EnergyPlus

The Windows build procedure is very similar to that for Linux and Mac OSX, using MSYS2 tools that you'll execute from a MSYS2 command window. However, some further adjustments are necessary as described below.

When you finish the build, try *TESP Demonstrations and Examples*.

6.3.1 Install Python Packages and Java

Download and install the 64-bit Miniconda installer, for Python 3.7 or later, from https://conda.io/miniconda.html. Install to c:Miniconda3, for easier use with MSYS2.

Then from a command prompt:

```
conda update conda
# tesp_support, including verification of PYPOWER dependency
pip install tesp_support --upgrade
opf
```

Download and install the Java Development Kit (11.0.5 suggested) from Oracle.

- for MSYS2, install to a folder without spaces, such as c:Javajdk-11.0.5
- the Oracle javapath doesn't work for MSYS2, and it doesn't find javac in Windows
- c:Javajdk-11.0.5bin should be added to your path

6.3.2 Set Up the Build Environment and Code Repositories

These instructions are based on https://github.com/gridlab-d/gridlab-d/blob/develop/BuildingGridlabdOnWindowsWithMsys2.docx For TESP, we're going to build with FNCS and HELICS, but not MATLAB or MySQL.

- Install a 64-bit version of MSYS2 from https://www.msys2.org. Accept all of the defaults.
- Start the MSYS2 environment from the Start Menu shortcut for "MSYS2 MSYS"

```
pacman -Syuu
```

- Enter y to continue
- When directed after a series of warnings, close the MSYS2 by clicking on the Close Window icon
- Restart the MSYS2 environment from the Start Menu shortcut for "MSYS2 MSYS"

```
pacman -Su
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
pacman -S --needed mingw-w64-x86_64-xerces-c
pacman -S --needed mingw-w64-x86_64-dlfcn
pacman -S --needed mingw-w64-x86_64-cmake
pacman -S --needed git jsoncpp
pacman -S --needed mingw64/mingw-w64-x86_64-zeromq
```

- Exit MSYS2 and restart from a different Start Menu shortcut for MSYS2 MinGW 64-bit
- You may wish to create a desktop shortcut for the 64-bit environment, as you will use it often

```
cd /c/
mkdir src
cd src
git config --global user.name "Your Name"
git config --global user.email "YourEmailAddress@YourDomain.com"
git clone -b feature/1173 https://github.com/gridlab-d/gridlab-d.git
git clone -b develop https://github.com/FNCS/fncs.git
git clone -b master https://github.com/GMLC-TDC/HELICS-src.git
git clone -b fncs-v8.3.0 https://github.com/FNCS/EnergyPlus.git
git clone -b develop https://github.com/pnnl/tesp.git
```

We're going to build everything to /usr/local in the MSYS2 environment. If you accepted the installation defaults, this corresponds to c:msys64usrlocal in the Windows environment. The Windows PATH should be updated accordingly, and we'll also need a GLPATH environment variable. This is done in the Windows Settings tool, choosing "Edit the system environment variables" or "Edit environment variables for your account" from the Settings search field.

- append c:\msys64\usr\local\bin to PATH
- append c:\msys64\usr\local\lib to PATH
- create a new environment variable GLPATH
- append c:\msys64\usr\local\bin to GLPATH

- append c:\msys64\usr\local\lib\gridlabd to GLPATH
- append c:\msys64\usr\local\share\gridlabd to GLPATH

Verify the correct paths to Java and Python for your installation, either by examining the PATH variable from a Windows (not MSYS) command prompt, or by using the Windows Settings tool. Insert the following to .bash_profile in your MSYS2 environment, substituting your own paths to Java and Python.

```
PATH="/c/Java/jdk-11.0.5/bin:${PATH}"
PATH="/c/Users/Tom/Miniconda3:${PATH}"
PATH="/c/Users/Tom/Miniconda3/Scripts:${PATH}"
PATH="/c/Users/Tom/Miniconda3/Library/mingw-w64/bin:${PATH}"
PATH="/c/Users/Tom/Miniconda3/Library/usr/bin:${PATH}"
PATH="/c/Users/Tom/Miniconda3/Library/bin:${PATH}"
```

The next time you open MSYS2, verify the preceeding as follows:

```
java -version
javac -version
python --version
python3 --version
```

6.3.3 Build FNCS and HELICS Link with GridLAB-D

ZeroMQ has already been installed with pacman to use with both FNCS and HELICS.

For FNCS, we still need to download CZMQ 4.1.1 source code from https://github.com/zeromq/czmq/releases We aren't prepared to deploy lz4 compression, and we have to specifiy custom libraries to link on Windows.

```
cd /c/src
tar -xzf czmq-4.1.1.tar.gz
cd czmq-4.1.1
// edit two lines of c:/msys64/mingw64/lib/pkgconfig/libzmq.pc so they read
//    Libs: -L${libdir} -lzmq -lws2_32 -liphlpapi -lpthread -lrpcrt4
//    Libs.private: -lstdc++
./configure --prefix=/usr/local --with-liblz4=no 'CXXFLAGS=-O2 -w -std=gnu++14' 'CFLAGS=-
↪O2 -w'
make
make install
```

Now build FNCS:

```
cd /c/src
cd fncs
autoreconf -if
./configure --prefix=/usr/local --with-czmq=/usr/local 'CXXFLAGS=-O2 -w -std=gnu++14'
↪'CFLAGS=-O2 -w'
make
make install
```

Use manual commands for the Java 11 FNCS Binding on Windows, because the Linux/Mac CMake files don't work on Windows yet. Also make sure that the JDK/bin directory is in your path.

```
cd /c/src/fncs/java
javac fncs/JNIfncs.java
jar cvf fncs.jar fncs/JNIfncs.class
javac -h fncs fncs/JNIfncs.java
g++ -DJNIfncs_EXPORTS -I"C:/Java/jdk-11.0.5/include" -I"C:/Java/jdk-11.0.5/include/win32
↪" -I/usr/local/include -I. -o fncs/JNIfncs.cpp.o -c fncs/JNIfncs.cpp
g++ -shared -o JNIfncs.dll fncs/JNIfncs.cpp.o "C:/Java/jdk-11.0.5/lib/jawt.lib" "C:/Java/
↪jdk-11.0.5/lib/jvm.lib" /usr/local/bin/libfncs.dll -lkernel32 -luser32 -lgdi32 -
↪lwinspool -lshell32 -lole32 -loleaut32 -luuid -lcomdlg32 -ladvapi32
```

To build HELICS 2.0 with Python and Java bindings:

```
cd /c/src/HELICS-src
mkdir build
cd build
cmake -G "MSYS Makefiles" -DCMAKE_INSTALL_PREFIX=/usr/local -DBUILD_SHARED_LIBS=ON -
↪DBUILD_PYTHON_INTERFACE=ON -DBUILD_JAVA_INTERFACE=ON -DJAVA_AWT_INCLUDE_PATH=NotNeeded␣
↪-DHELICS_DISABLE_BOOST=ON -DCMAKE_BUILD_TYPE=Release ..
make
make install
```

Test that HELICS and FNCS start:

```
helics_player --version
helics_recorder --version
fncs_broker --version # look for the program to start, then exit with error
```

Finally, build and test GridLAB-D with FNCS. If you encounter build errors with GridLAB-D, please try adding *-std=c++11* to *CXXFLAGS*.

```
cd /c/src/gridlab-d
autoreconf -isf
./configure --build=x86_64-mingw32 --with-fncs=/usr/local --with-helics=/usr/local --
↪prefix=/usr/local --with-xerces=/mingw64 --enable-silent-rules 'CXXFLAGS=-O2 -w -
↪std=gnu++14' 'CFLAGS=-O2 -w' 'LDFLAGS=-O2 -w -L/mingw64/bin'
make
make install
gridlabd --validate
```

In order to run GridLAB-D from a regular Windows terminal, you have to copy some additional libraries from c:\msys64\mingw64\bin to c:\msys64\usr\local\bin. This step must be repeated if you update the gcc compiler or ZeroMQ library.s

- libdl.dll
- libgcc_s_seh-1.dll
- libsodium-23.dll
- libstdc++-6.dll
- libwinpthread-1.dll
- libzmq.dll

6.3.4 Build EnergyPlus

Install the archived version 8.3 from https://github.com/NREL/EnergyPlus/releases/tag/v8.3.0 We need this for some critical support files that aren't part of the FNCS-EnergyPlus build process. Copy the following from c:\EnergyPlusV8-3-0 to c:\msys64\usr\local\bin:

- Energy+.idd
- PostProcess\ReadVarsESO.exe

From the MSYS2 terminal:

```
cd /c/src/energyplus
mkdir build
cd build
cmake -G "MSYS Makefiles" -DCMAKE_INSTALL_PREFIX=/usr/local ..
make
make install
```

The Makefiles put energyplus.exe and its DLL into /usr/local. You have to manually copy the following build products from /usr/local to /usr/local/bin:

- energyplus.exe
- energyplusapi.dll

6.3.5 Build eplus_agent

From the MSYS2 terminal

```
cd /c/src/tesp/src/energyplus
cp Makefile.win Makefile
cp config.h.win config.h
make
make install
```

6.3.6 Build ns3 with HELICS

```
cd /c/src
git clone https://gitlab.com/nsnam/ns-3-dev.git
cd ns-3-dev
git clone https://github.com/GMLC-TDC/helics-ns3 contrib/helics
./waf configure --check-cxx-compiler=g++ --with-helics=/usr/local --disable-werror --
↪enable-examples --enable-tests
./waf build
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] R. Lincoln. (2017). PYPOWER. Available: https://pypi.python.org/pypi/PYPOWER.

[2] D. G. Holmberg, D. Hardin, R. Melton, R. Cunningham, and S. Widergren, "Transactive Energy Application Landscape Scenarios," Smart Grid Interoperability Panel2016.

[3] ANSI, "ANSI C84.1-2016; American National Standard for Electric Power Systems and Equipment—Voltage Ratings (60 Hz)," ed, 2016.

[4] ASHRAE, "ANSI/ASHRAE standard 55-2010 : Thermal Environmental Conditions for Human Occupancy," 2010.

[5] J. K. Kok, C. J. Warmer, and I. G. Kamphuis, "PowerMatcher: Multiagent Control in the Electricity Infrastructure," presented at the Proceedings of the Fourth International joint conference on Autonomous agents and multiagent systems, The Netherlands, 2005.

[6] TeMix Inc. (2017). TeMix. Available: http://www.temix.net.

[7] Ila Arlow, Jim; Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, 2nd edition, June 2005.

[8] David P Chassin, Jason C Fuller, and Ned Djilali. GridLAB-D: An Agent-Based Simulation Framework for Smart Grids. *Journal of Applied Mathematics*, 2014:1–12, 2014.

[9] Jason Fuller, Kevin P Schneider, and David P Chassin. Analysis of Residential Demand Response and Double-Auction Markets. In *Power and Energy Society General Meeting, 2011 IEEE*, 1–7. February 2011. URL: https://ieeexplore.ieee.org/document/6039827/, doi:10.1109/PES.2011.6039827.

[10] Donald J Hammerstrom, Charles D Corbin, N Fernandez, J S Homer, Atefe Makhmalbaf, Robert G Pratt, Abhishek Somani, E I Gilbert, Shawn A Chandler, and R Shandross. Valuation of Transactive Systems. Technical Report PNNL-25323, Pacific Northwest National Laboratory, Richland, WA, May 2016. URL: https://bgintegration.pnnl.gov/pdf/ValuationTransactiveFinalReportPNNL25323.pdf.

[11] Sarmad Hanif, Laurentiu Marinovici, Mitch Pelton, Trevor Hardy, and Thomas E. McDermott. Simplified transactive distribution grids for bulk power system mechanism development. In *2021 IEEE Power Energy Society General Meeting (PESGM)*, 01–05. July 2021. doi:10.1109/PESGM46819.2021.9638030.

[12] He Hao, Charles D Corbin, Karanjit Kalsi, and Robert G Pratt. Transactive Control of Commercial Buildings for Demand Response. *Power Systems, IEEE Transactions on*, 32(1):774–783, 2017. URL: http://ieeexplore.ieee.org/document/7460977/, doi:10.1109/TPWRS.2016.2559485.

[13] Qiuhua Huang, Tom McDermott, Yingying Tang, Atefe Makhmalbaf, Donald Hammerstrom, Andy Fisher, Laurentiu Marinovici, and Trevor D Hardy. Simulation-Based Valuation of Transactive Energy Systems. *Power Systems, IEEE Transactions on*, pages 1–1, May 2018. URL: https://ieeexplore.ieee.org/document/8360969/, doi:10.1109/TPWRS.2018.2838111.

[14] IEEE. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification - Redline. *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, pages 1–378, Aug 2010. doi:10.1109/IEEESTD.2010.5954120.

[15] IEEE. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, Aug 2010. doi:10.1109/IEEESTD.2010.5553440.

[16] IEEE. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Object Model Template (OMT) Specification. *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)*, pages 1–110, Aug 2010. doi:10.1109/IEEESTD.2010.5557731.

[17] IEEE. Ieee guide for electric power distribution reliability indices. *IEEE Std 1366-2012 (Revision of IEEE Std 1366-2003)*, pages 1–43, May 2012. doi:10.1109/IEEESTD.2012.6209381.

[18] IEEE. Ieee guide for collecting, categorizing, and utilizing information related to electric power distribution interruption events. *IEEE Std 1782-2014*, pages 1–98, Aug 2014. doi:10.1109/IEEESTD.2014.6878409.

[19] NIST. NIST Transactive Energy Challenge. 2017. URL: https://pages.nist.gov/TEChallenge/.

[20] Bryan Palmintier, Dheepak Krishnamurthy, Philip Top, Steve Smith, Jeff Daily, and Jason Fuller. Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework. In *2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, 1–6. IEEE, 2017. URL: http://ieeexplore.ieee.org/document/8064542/, doi:10.1109/MSCPES.2017.8064542.

[21] Kevin P Schneider, Yousu Chen, David P Chassin, Robert G Pratt, David W Engel, and Sandra E Thompson. Modern grid initiative distribution taxonomy final report. 11 2008. URL: https://www.osti.gov/biblio/1040684, doi:10.2172/1040684.

[22] Steven E Widergren, Donald J Hammerstrom, Qiuhua Huang, Karanjit Kalsi, Jianming Lian, Atefe Makhmalbaf, Thomas E McDermott, Deepak Sivaraman, Yingying Tang, Arun Veeramany, and James C Woodward. Transactive Systems Simulation and Valuation Platform Trial Analysis. Technical Report PNNL-26409, Pacific Northwest National Laboratory (PNNL), Richland, WA (United States), Richland, WA, April 2017. URL: http://www.osti.gov/servlets/purl/1379448/, doi:10.2172/1379448.

[23] Haifeng Zhang, Yevgeniy Vorobeychik, Joshua Letchford, and Kiran Lakkaraju. Data-driven agent-based modeling, with application to rooftop solar adoption. *Autonomous Agents and Multi-Agent Systems*, 30(6):1023–1049, 2016. URL: https://doi.org/10.1007/s10458-016-9326-8, doi:10.1007/s10458-016-9326-8.

[24] Gridwise Architecture Council. GridWise Transactive Energy Framework Version 1.1. Technical Report, Gridwise Architecture Council, July 2019. URL: https://www.gridwiseac.org/pdfs/pnnl_22946_gwac_te_framework_july_2019_v1_1.pdf.

[25] H. Li and L. Tesfatsion. The AMES Wholesale Power Market Test Bed: A Computational Laboratory for Research, Teaching, and Training. In *2009 IEEE Power Energy Society General Meeting*, 1–8. July 2009. doi:10.1109/PES.2009.5275969.

[26] K. P. Schneider, Y. Chen, D. Engle, and D. Chassin. A Taxonomy of North American Radial Distribution Feeders. In *2009 IEEE Power Energy Society General Meeting*, 1–6. July 2009. doi:10.1109/PES.2009.5275900.

[27] V. Sultan, B. Alsamani, N. Alharbi, Y. Alsuhaibany, and M. Alzahrani. A predictive model to forecast customer adoption of rooftop solar. In *2016 4th International Symposium on Computational and Business Intelligence (ISCBI)*, 33–44. Sep. 2016. doi:10.1109/ISCBI.2016.7743256.

[28] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas. Matpower: steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Transactions on Power Systems*, 26(1):12–19, Feb 2011. doi:10.1109/TPWRS.2010.2051168.

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z