

# Switch 2.0 Tutorial: Pre-Tutorial Setup

Matthias Fripp  
Assoc. Prof. of Electrical Engineering  
University of Hawaii, Manoa  
mfripp@hawaii.edu

Before you come for the tutorial, please complete the steps described here to install Switch on a laptop computer and learn some background information. Section 2.2 and 2.3 are optional. You will need an Internet connection while you do these. Required steps are marked by a blue highlight in the margin like this paragraph. Explanatory text and optional steps don't have this mark.

## 1 Software setup (approx. 1 hour; requires 4-5 GB disk space)

This section describes how to install Switch on your computer so it is ready to solve power system planning problems. Switch depends on a collection of mostly open-source software:

- **Switch core modules** and **your extension modules** define a power system optimization model in relatively easy-to-understand Python code.
- **Switch** loads the modules you have selected and the corresponding data, constructs an optimization model, solves it and reports results.
- **Pyomo** is a general-purpose optimization modeling framework for Python. Switch uses this package to define the elements of your optimization model and call a solver.
- Pyomo converts the Switch model into a standardized, computer-readable form and sends it to an **external solver** (e.g., glpk, cbc, cplex or gurobi). The external solver does the intense computation required to find an optimal plan.
- **Anaconda** provides an easy, standardized way to install Python, Pyomo, glpk and other tools on any computer system.

The instructions below will show you how to setup the following:

- Anaconda Python environment
- Switch and software it depends on (Pyomo and glpk)
- Example models to use in this tutorial
- Atom text editor

All of these tools are open-source and cross-platform, so you should be able to use them on any computer.

Disk usage: Switch itself is quite small, but it depends on the Anaconda distribution (0.3-0.5 GB), and a number of Python packages (0.5-1.5 GB). We will also use the Atom text editor for this tutorial (0.8 GB), and install some tutorial data (0.3 GB). Note that the higher disk usage numbers are for Windows installations.

### 1.1 Installing Anaconda and Python

Download and install the “mini” version of Anaconda from <https://docs.conda.io/en/latest/miniconda.html>. I recommend selecting the 64-bit Python 3+ version (other versions will probably work too). On a Mac, the “.pkg” installer is easier to use than the “bash” one. Note that you do not need administrator privileges to install Anaconda or add packages to it if you choose the “Install for: Just Me” option during installation.

(You can also use the “full” version of Anaconda from <https://www.anaconda.com/distribution/>, but that requires 2–3 GB more disk space).

### 1.2 Installing Switch, Pyomo and glpk

Open Terminal.app (OS X: in Applications/Utilities folder) or an Anaconda command prompt (Windows: Start > Anaconda3 > Anaconda Command Prompt). Then type this command and press Enter or return:

```
conda install -c conda-forge switch_model
```

Follow the prompts to install Switch and related software.

You can check that everything has been installed by running each of these commands:

```
glpsol --version  
python --version  
pyomo --version  
switch --version
```

You should see version numbers for each one, and no errors.

### 1.3 Retrieving tutorial data and Switch source code

Open Terminal.app or an Anaconda command prompt if you don’t have it open already. Then type this command and press Enter or return:

```
conda install git
```

Follow the prompts to install “git”. On a Mac, if you are prompted to install the command-line developer tools, click “Install”.

Use the “cd” and “mkdir” commands navigate to a good location to install the tutorial data (e.g., run “cd Documents” then “mkdir tutorial” then “cd tutorial”, and the data will be placed inside a “tutorial” folder inside your “Documents” folder). Then use the following command to copy the tutorial data to a subdirectory called “switch\_tutorial” within the current directory:

```
git clone --depth=1 https://github.com/switch-model/switch_tutorial.git
```

Now use the following command to copy the source code for Switch into a subdirectory called “switch” within the current directory. This will be useful for inspecting the source code later.

```
git clone --depth=1 https://github.com/switch-model/switch.git
```

Note: you now have two copies of Switch on your system—one for running and one for viewing. It is possible to use one copy for both, but it is a little tricky. If you want to do that, please see the last section of <https://github.com/switch-model/switch/blob/master/INSTALL.md>.

#### 1.4 Installing Atom text editor and tablr extension

For this tutorial, I assume you are using the Atom text editor to view and edit code and data files. You can use a different text editor if you like, but it should be capable of doing programming-oriented tasks, like quickly adjusting the indentation of groups of lines in a text file. If you prefer, you can also open the .csv data files directly in your spreadsheet software instead of using Atom and the tablr extension.

Download and install the Atom text editor from <https://atom.io>.

**On a Mac**, you should finish the installation by copying the Atom app from your download folder to your Applications folder.

**On Windows**, Atom uses a standard installer, but it may spend a few minutes on the animated installer screen.

If you need more info on installing Atom, see <https://flight-manual.atom.io/getting-started/sections/installing-atom/>.

If you'd like a quick intro to the program, see <https://flight-manual.atom.io/getting-started/sections/atom-basics/>.

Next, open Atom from the Start menu or Applications folder. View and/or close the introductory tabs to clean up the display a little.

Now you will install the “atom-tablr” package, which allows you to easily view and edit tabular text files, such as the .csv files used by Switch. This step is optional, but it will make the .csv files much easier to read.

The atom-tablr package is great, but unfortunately slightly broken and no longer maintained. However, you can install a fixed version as follows: In Atom, choose the Atom > Preferences

menu (Mac) or File > Settings menu (Windows). Then click on the “+ Install” tab. In the “Search packages” box at the top, type “https://github.com/mfripp/atom-tablr.git” (without the quotes). Below this, you should see a box that says “git+https://github.com/mfripp/atom-tablr.git”. Click on the “Install” button in this box. It should take a minute or so to install, and then the “Install” button will change to “Settings”, “Uninstall” and “Disable”. (If this doesn’t work, see footnote 1.)

Next, I recommend clicking on the “Settings” button for atom-tablr and adjusting the following settings:

Csv Editor (about 1/4 way down): turn on “File Header”

Table Editor (about 1/2 way down): set Column Width to 180

(On some systems the Settings button does not appear; if so, you can skip this step.)

By default, Atom uses a light-on-dark color scheme. If you’d like to change this, you can do that via the Themes tab in the Settings pane (Atom > Preferences > Themes on a Mac, File > Settings > Themes on Windows).

Optional (not used in this tutorial): For the future, I also recommend installing the “hydrogen” and “hydrogen-python” packages. These allow you to run Python code directly from Atom, which is great for testing and troubleshooting scripts as you write them (e.g., data conversion and analysis scripts). To install those, go back to Atom and open Atom > Preferences... or File > Preferences.... Choose the Install tab. Search for “hydrogen”. Click the “Install” buttons for “hydrogen” and “hydrogen-python”. To support these, you will also need to run these two commands (one time) in a Terminal window or Anaconda Command Prompt: “conda install ipykernel”, then “python -m ipykernel install --user”. You may also be interested in the Atom packages for running R, Stata or LaTeX code from the editor.

### 1.5 Installing a commercial solver (optional)

The conda command automatically installs the open-source glpk solver along with Switch. This and other open-source solvers (e.g., cbc) are able to solve small test cases, but are not fast enough to solve large models. So I generally use proprietary solvers (cplex or gurobi) in practice. These are expensive for professional use, but it is possible to get a trial license before you buy a long-term one. Academics can also get full licenses for free. Note that it is important to get a license (temporary or long-term) for the *full* version of the software, not the free or community version that only supports small problem sizes.

You can complete this tutorial using only glpk, but if you would like to experiment on your own with the larger models, you should install cplex or gurobi. You can obtain licenses and download these solvers from here:

---

<sup>1</sup> This sometimes fails on Windows. In that case, open a non-Anaconda command prompt (you can find it by searching for cmd.exe in the search bar), then run the command “apm install https://github.com/mfripp/atom-tablr.git”. After it runs, restart Atom, go to File > Settings, then the Packages tab, then search for tablr among the installed packages, and continue from there.

Professional:

<https://www.gurobi.com/products/gurobi-optimizer/>

<https://www.ibm.com/products/ilog-cplex-optimization-studio/pricing>

Academic:

<https://www.gurobi.com/academia/>

<https://developer.ibm.com/docloud/blog/2019/07/04/cplex-optimization-studio-for-students-and-academics/>

Once you have installed these, you can test that they are available by running one of these pairs of commands from a command prompt or terminal window:

Gurobi on Windows:

```
gurobi.bat  
exit()
```

Gurobi on Mac or Linux

```
gurobi.sh  
exit()
```

CPLEX

```
cp1ex  
quit
```

## 2 Introduction to Switch, Pyomo and Python (0.5 – 9 hours)

This section points you to some useful, quick introductions to Switch, Python and Pyomo. The latter two are optional, but recommended if you will be using Switch extensively or defining custom behaviors (new technologies, rules or policies).

### 2.1 Introduction to Switch (0.5 – 2 hours)

For a quick overview of Switch, please read Section 2 of the paper on Switch 2.0 at <https://doi.org/10.1016/j.softx.2019.100251>.

The following are optional: You can read section 3 of the Switch paper for an overview of the case study we'll examine during this tutorial. And if you would like more detail on Switch, please see the Supplementary Material for the Switch 2.0 paper at <https://ars.els-cdn.com/content/image/1-s2.0-S2352711018301547-mmc1.pdf>.

## 2.2 Introduction to Python (optional, 1 – 6 hours)

I recommend reading sections 3 and 4 of the Python introduction at <https://docs.python.org/3/tutorial/>. If you would like a deeper understanding, sections 5 and 6 are also worth reading.

If you want to run sample code from the Python tutorial, you can do so as follows: Open Terminal.app (Mac) or an Anaconda command prompt (Windows). Then type “python<enter>” to start the Python interpreter. Then copy code from the Python tutorial into the interpreter. Generally the code marked with “>>>” or “...” is code that you can type to the Python interpreter. But you shouldn’t copy the >>> or ... symbols themselves.

## 2.3 Introduction to Optimization and Pyomo (optional, 1 – 3 hours)

I recommend going through the following sections at <https://pyomo.readthedocs.io/> to get an introduction to Pyomo, the optimization software used by Switch.

- [Pyomo Overview](#)
- [Pyomo Modeling Components](#)
- [Solving Pyomo Models](#)

This will enable you to read and write Switch code, which is just Pyomo code applied to power system modeling. i.e., a Switch model is a Pyomo AbstractModel used to optimize the design of a power system.

**Notation:** In the Pyomo introduction, you will see problems with a notation like this:

$$\begin{aligned} \min \quad & c_1x_1 + c_2x_2 \\ \text{s.t.} \quad & \\ & a_{11}x_1 + a_{12}x_2 \geq b_1 \\ & a_{21}x_1 + a_{22}x_2 \geq b_2 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{aligned}$$

This is a common way to describe mathematical optimization problems. It means "find values for  $x_1$  and  $x_2$  that will minimize the value of  $c_1x_1 + c_2x_2$ , such that all the specified constraints are satisfied."

In this problem, the  $x$  values are called **decision variables** (these are numbers that will be chosen when the problem is run, e.g., the amount of power to produce from a project during a particular hour), the  $a$ ,  $b$ , and  $c$  values are **parameters** (data you know when you set up the problem, e.g., the maintenance cost per MWh produced from a project),  $c_1x_1 + c_2x_2$  is the **objective function** (the value to be minimized or maximized), and the other equations are the **constraints** (e.g., that power output is less than or equal to installed capacity).

Now you are all set for the tutorial. See you next week.

# Switch 2.0 Tutorial

Environmental Defense Fund  
Emerging Economics Issues Workshop  
New York, New York  
August 22, 2019

Matthias Fripp  
Assoc. Prof. of Electrical Engineering  
University of Hawaii, Manoa  
mfripp@hawaii.edu

Before you come for the tutorial, please complete the steps described in the “Pre-Tutorial Setup” document (sent previously) to install Switch on a laptop computer and learn some background information.

If you completed those steps before Tuesday afternoon, please also update to the latest version of the tutorial data as follows: go to a Terminal window or Anaconda Command Prompt, navigate to the tutorial\_data directory (possibly “cd Documents\tutorial\switch\_tutorial” on Windows or “cd Documents/tutorial/switch\_tutorial” on other platforms), then run “git pull”. This will retrieve the latest version of this repository.

In this tutorial, required steps are marked by a blue highlight in the margin like the paragraph above. Explanatory text and optional steps don’t have this mark.

## 1 Basic modeling with Switch

### 1.1 Where to find documentation

Before we start, here are some places you can look for documentation on Switch.

- We are currently working on a website to draw all these resources together and provide cross-links between model logic and input data tables. That will be available at <http://switch-model.org>. For now, this provides shortcuts to the documentation listed below.
- The Switch 2.0 paper at <https://doi.org/10.1016/j.softx.2019.100251> provides a concise overview of Switch 2.0 and is recommended for all users.
- Section 4 of the Supplementary Material for the Switch 2.0 paper at <https://ars.els-cdn.com/content/image/1-s2.0-S2352711018301547-mmc1.pdf> provides a complete mathematical description of Switch. This describes the logic of all the Switch modules and also identifies the decision variables, indexing sets and model parameters (input data) defined or used in each module.
- Running “switch --help” or “switch solve --help” (in a model directory) from the command line will give you descriptions of all the available command-line options.
- The Switch source code provides another view of the elements described in the Supplementary Material, i.e., the model code itself. Each module includes extensive comments, and after some practice you will find that these are a great reference for which model component is used for what purpose, and how the components are

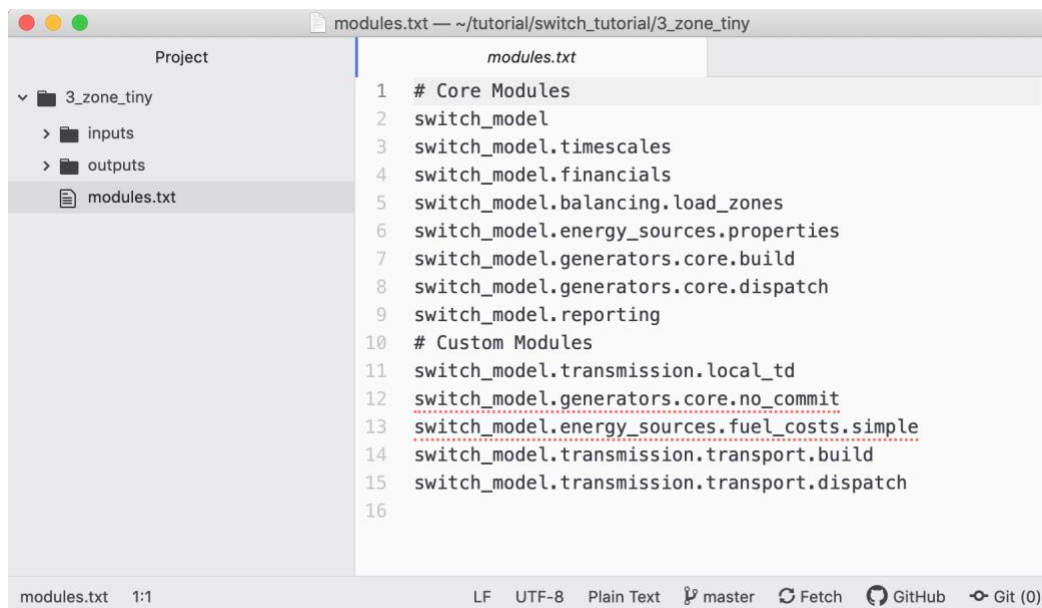
defined. For example, you can use shift-control-F/shift-command-F in Atom to find all references to a particular component. It is also possible to identify which columns need to be defined in each input table by reading the `load_inputs` section at the bottom of each module. These modules can also serve as a template for writing your own custom modules (which will look and work just like standard Switch modules). There are two main ways to view Switch source code:

- Browse the Switch repository at <https://github.com/switch-model/switch>
- Download a copy of the source code and open that in a text editor. This is the method we'll use in this tutorial.
- One of the hardest parts of working with Switch is assembling and formatting all the inputs for a complete, working model of your power system. Rather than working directly from the model description, it is often easier to refer to a pre-existing dataset and replicate that structure with your own data. The example datasets we'll use for this tutorial should give you a good start. You can find additional examples in the "examples" directory of the Switch repository at <https://github.com/switch-model/switch> or in the Switch-Hawaii repository at <https://github.com/switch-hawaii/>.

## 1.2 Model configuration: `modules.txt`

In the pre-tutorial setup, you copied data to a folder called "switch\_tutorial". Now we will work with the simple model called "3\_zone\_tiny" inside this folder.

Open Atom, then choose File > Open... (Mac) or File > Open Folder... (Windows) and select the `3_zone_tiny` folder (not the files inside, just the folder). Then click on `modules.txt`. This should create a window similar to the one below.



```
modules.txt — ~/tutorial/switch_tutorial/3_zone_tiny
Project
  3_zone_tiny
    inputs
    outputs
    modules.txt
modules.txt
1 # Core Modules
2 switch_model
3 switch_model.timescales
4 switch_model.financials
5 switch_model.balancing.load_zones
6 switch_model.energy_sources.properties
7 switch_model.generators.core.build
8 switch_model.generators.core.dispatch
9 switch_model.reporting
10 # Custom Modules
11 switch_model.transmission.local_td
12 switch_model.generators.core.no_commit
13 switch_model.energy_sources.fuel_costs.simple
14 switch_model.transmission.transport.build
15 switch_model.transmission.transport.dispatch
16
modules.txt 1:1  LF UTF-8 Plain Text master Fetch GitHub Git (0)
```



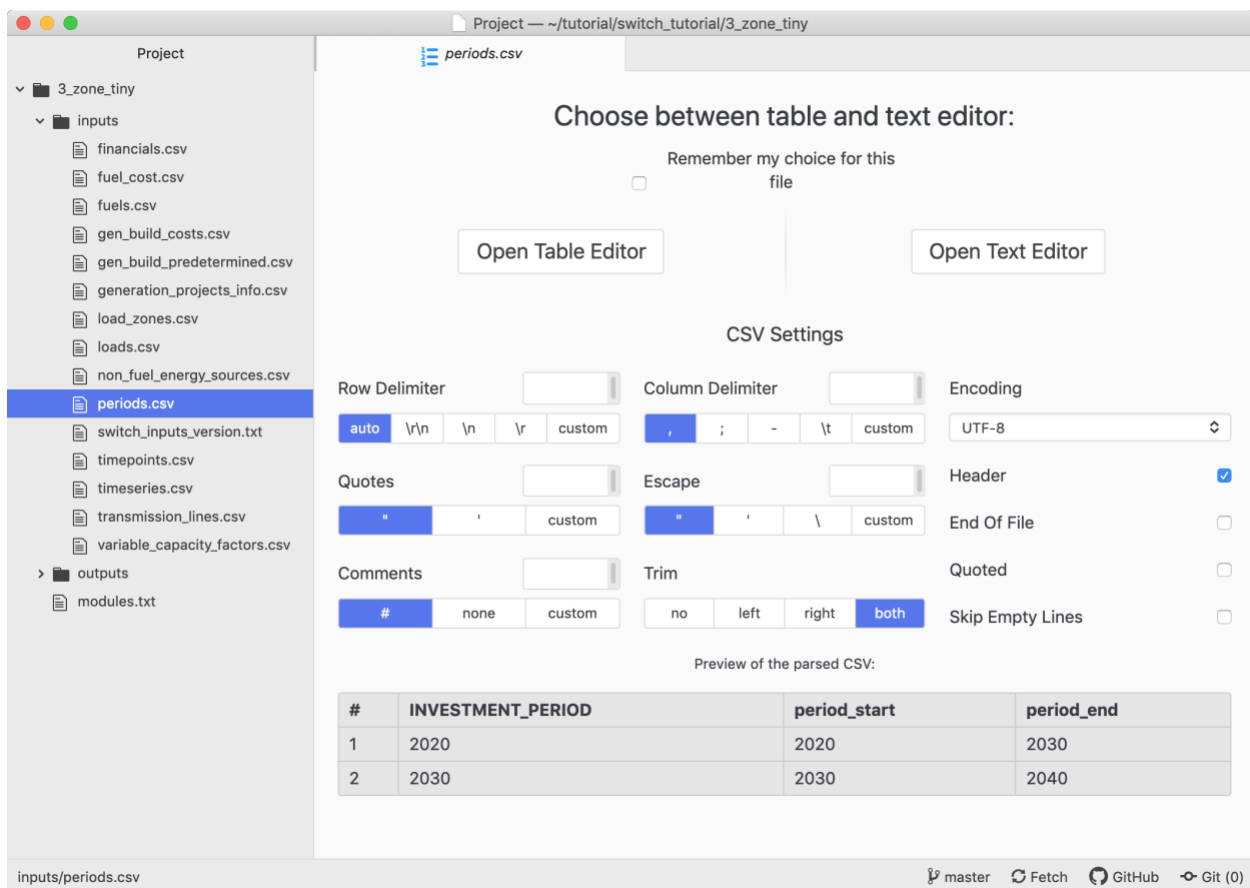
**modules.txt** lists all the Python modules that will be used to define your model. Some of these are required for every model: `timescales`, `financials`, `load_zones`, `energy_sources.properties`, `generators.core.build`, `generators.core.dispatch`, `generators.core.no_commit` (or an alternative) and `energy_sources.fuel_costs.markets` (or an alternative). The rest are optional, and give additional behavior or rules. Most people use at least the reporting module, which saves the values of all decision variables (all the choices made by Switch). You can write comments in this file (or turn active lines into comments) by putting a “#” sign at the start.

## 1.3 Model inputs

### 1.3.1 Opening .csv files in Atom

For this tutorial, we will view and edit .csv files in Atom, using the `tablr` extension you installed earlier. If you prefer, you can open .csv files directly in Excel or LibreOffice. You can use the File > Open dialog in Excel or LibreOffice to find the files, or on most computers you can double-click on the .csv files in Windows Explorer or Finder and they will open.

Click on the “inputs” folder to look at the files inside. Then click on “periods.csv”. You should see a window similar to the one below.



Click on “Open Table Editor”, and you will get a window like this:

	INVESTMENT_PERIOD	period_start	period_end
1	2020	2020	2030
2	2030	2030	2040

You can use similar steps for the other .csv files.

### 1.3.2 Model input files

Using the procedure above, we will look at several input files.

All of the input files have a similar structure. The leftmost column(s) contain index entries, then the columns to the right of those contain parameter values. Usually the index column names are written with all capital letters, and the parameter names are written in lower case. Each row in the input file corresponds to one entity, and the index values identify which entity (e.g., which investment period). Then the parameter fields provide data related to that entity (e.g., when an investment period starts or ends). Note that omitted values are given as periods (“.”). Each index value is part of a set, e.g., the set of all possible investment periods. In some cases, those sets are defined internally by Switch, and in some cases they are read directly from the first column(s) of an input file.

**periods.csv** defines the investment calendar for your model. Assets can be built, expanded or retired at the start of each period. The first column (labeled INVESTMENT\_PERIOD) defines the set PERIODS in Switch. These are the labels that are always used to refer to these blocks of time. Then the period\_start and period\_end parameters define the start and end of each period. (There is some ambiguity about whether period\_end should be the last full year of a period, or the point in time when the period ends, i.e., the start year of the next period. Switch will accept either and makes an intelligent guess about how you’ve set it up.)

**timeseries.csv** defines all the timeseries (independent time samples) that will be used in your model. There is one row for each timeseries. Each timeseries is a collection of chronologically linked timepoints that falls within a particular investment period. e.g., for many models, each timeseries will represent one sample day, and it will contain 24 1-hour timepoints or 12 2-hour timepoints. The first column in timeseries.csv is the name of the timeseries (used to define Switch’s TIMESERIES set), then there are columns to define which investment period the timeseries falls in, how many hours are spanned by each timepoint in the timeseries and how many timepoints there are in each timeseries. Finally, ts\_scale\_to\_period shows a weight to

apply to each timeseries within the investment period. When calculating costs, energy use, emissions, etc., Switch treats each timeseries as if it occurred this many times during the corresponding investment period (uniformly spread throughout the period). If you calculate  $ts\_duration\_of\_tp * ts\_num\_tps * ts\_scale\_to\_period$  for each timeseries, then sum across all the timeseries in each period, it should add up to the duration of the period in hours (Switch will give you an error message if it doesn't). This framework allows you to give more weight to the most common days, but also include some rare, hard-to-serve days, with appropriate weighting. Usually there are multiple timeseries for each period, and usually you would use the same values of  $ts\_duration\_of\_tp$  and  $ts\_num\_tps$  for all timeseries. However, for speed and illustration, this model includes relatively few timeseries and varies the density of timepoints and duration of timeseries.

**timepoints.csv** defines all the timepoints used in Switch. The first column lists the names of all the timepoints in the model (i.e., it defines the TIMEPOINTS set). These can be numbers or text. The timestamp column is optional and contains a human-readable label for each timepoint which is used in reports. The final column identifies which timeseries each timepoint belongs to.

In Switch, operational decisions (generator commitment and dispatch, load balancing, etc.) are calculated for each timepoint. Collections of sequential timepoints form timeseries, typically representing a single sample day of weather and load conditions. Switch models the timepoints in each timeseries circularly, as if the last timepoint leads back to the first one. This is equivalent to modeling each timeseries as an infinite sequence of identical days when the system must be able to run successfully (so, e.g., storage must return to its original state by the end of the day). This sequential linkage enables consideration of time-linked factors like minimum up- or down-time for generators or avoiding overcharging or undercharging of storage. Then there are several, independent timeseries within each investment period. Investment/construction decisions are made at the start of each period, and equipment is then operated over all the timeseries and timepoints within that period.

All the time-related inputs are read by the `switch_model.timescales` module.

**load\_zones.csv** defines the topology of the power system. For this model, we are using a ball-and-spoke model—the most common approach used with Switch—where power can be moved freely within each load zone, but there may be congestion between load zones. Even for copperplate models, information must be provided for the single load zone used for all assets. The first column of `load_zones.csv` defines the LOAD\_ZONES set. “dbid” is an optional column that can be used to cross-reference the load zone with your database. The `existing_local_td` and `local_td_annual_cost_per_mw` columns are used by the optional `switch_model.transmission.local_td` module to estimate the cost of upgrading local transmission and distribution within each load zone, based on the peak load served each year.

**loads.csv** specifies the amount of load to serve (in MW) in each load zone during each timepoint. Note that LOAD\_ZONE and TIMEPOINT are index columns, and the last column is the load.

**generation\_projects\_info.csv** defines all possible generation projects. Each project has a unique ID, and these define the GENERATION\_PROJECTS set in Switch. Generation projects define Switch's construction options—they can be a single project that could be built, but often they represent a stack of identical projects that can be built in that load zone, and Switch decides how much capacity (in MW) to develop from this stack. This includes the following columns:

Parameter	Description
GENERATION_PROJECT	name of this generation project; defines the GENERATION_PROJECTS set in Switch
gen_tech	general class of technology for each project (used for reporting and/or users' custom modules)
gen_load_zone	load zone where the project(s) would be connected
gen_connect_cost_per_mw	capital cost to connect to the transmission network
gen_capacity_limit_mw	(optional) shows amount of capacity that can be built in this project or stack of projects. This is most often used to reflect limits on available resources for renewable energy projects, but can also limit thermal plants. If not specified, there is no limit.
gen_full_load_heat_rate	full-load heat rate for thermal plants, in million Btu per MWh; should be omitted for renewable projects
gen_variable_om	O&M cost per MWh produced
gen_max_age	age limit for projects of this type; costs are amortized over this period, and capacity is automatically retired at this age (Switch may then decide to build identical, new capacity to replace it)
gen_min_build_capacity	(optional) minimum size to build if the project is developed; if specified, the developed capacity for the project can be 0 MW or $\geq$ gen_min_build_capacity.
gen_scheduled_outage_rate	(optional) scheduled maintenance outage rate; used to derate baseload generation capacity (generally only suitable for large power systems)
gen_forced_outage_rate	(optional) forced outage rate; used to derate available capacity from all generators at all times (generally only suitable for large power systems)
gen_is_variable	identifies variable generators such as wind or solar projects, which have availability specified in the variable_capacity_factors.csv file (0 or 1 value)

<code>gen_is_baseload</code>	identifies baseload power plants, which must run at a constant output level (0 or 1 value)
<code>gen_is_cogen</code>	(optional) not implemented in standard modules, but available for custom modules
<code>gen_energy_source</code>	name of energy source (fuel or renewable resource) used by this project; can also be “multiple”, in which case a list of fuels must be provided in <code>gen_multiple_fuels.dat</code>
<code>gen_unit_size</code>	(optional) fixed size for incremental capacity additions or unit commitment; has no effect unless <code>generators.core.gen_build_discrete</code> and/or <code>generators.core.commit.discrete</code> modules are loaded
<code>gen_ccs_capture_efficiency</code>	(optional) fraction of CO <sub>2</sub> emissions captured by CCS equipment
<code>gen_ccs_energy_load</code>	(optional) fraction of power production lost to CCS operation
<code>gen_storage_efficiency</code>	(optional) round-trip efficiency for storage projects
<code>gen_store_to_release_ratio</code>	(optional) storage power rating for a storage project, given as a ratio vs. the power output rating
<code>gen_storage_energy_to_power_ratio</code>	(optional) fixed ratio of storage energy (MWh) per unit of capacity (MW) installed, i.e., number of hours of storage; Switch will optimize the number of hours of storage if this is not specified
<code>gen_storage_max_cycles_per_year</code>	(optional) limit on number of full charge/discharge cycles per year for storage projects
<code>gen_min_uptime</code>	(optional) minimum number of hours that a generator must remain committed (turned on) after it is first committed
<code>gen_min_downtime</code>	(optional) minimum number of hours that a generator must remain uncommitted after it is first decommitted (turned off)
<code>gen_startup_fuel</code>	(optional) amount of fuel needed to startup this generator (millions of Btu per MW of capacity started up)

**gen\_build\_costs.csv** shows the construction cost (`gen_overnight_cost`, \$/MW) and fixed O&M (`gen_fixed_om`, \$/MW-year) for all periods when capacity can be added to each generation project, as well as in the years prior to the study when existing plants were built. It also shows the capital cost of expanding the energy capacity for storage projects if they are used (`gen_storage_energy_overnight_cost`, \$/MWh). Entries in this file determine when new capacity can be built: project capacity can be expanded in a particular period if and only if a cost is given in `gen_build_costs.csv` for that project in that period.

**gen\_build\_predetermined.csv** lists all capacity that was added to generation projects during years (build\_year) before the study began (gen\_predetermined\_cap, in MW). Capacity specified here will automatically be retired after it reaches max\_age\_years.

**fuel\_cost.csv** identifies the fuels available in each load zone during each study period, and the cost of each fuel in dollars per million Btu. This works with the switch\_model.energy\_sources.fuel\_costs.simple module. Alternatively, you can use switch\_model.energy\_sources.fuel\_costs.markets and define fuel supply curves in fuel\_supply\_curves.csv (this is done in the model in section 2).

**fuels.csv** defines the list of possible fuels for your model (this is Switch's FUELS set). It can optionally show the direct CO<sub>2</sub> intensity of each (tonnes CO<sub>2</sub> per million Btu combusted), upstream CO<sub>2</sub> intensity and rps eligibility of each fuel (1 or 0).

**non\_fuel\_energy\_sources.csv** lists all energy sources that aren't in fuels.csv, e.g., names of renewable resources. (This defines Switch's NON\_FUEL\_ENERGY\_SOURCES set.) The gen\_energy\_source in generation\_projects\_info.csv must be in either FUELS or NON\_FUEL\_ENERGY\_SOURCES.

**financials.csv** defines financial parameters for the model. All costs are discounted to the base\_financial\_year using the discount\_rate. Capital costs are amortized over the life of the project using the interest\_rate (optional, defaults to discount\_rate). Note that unlike the other parameters we have viewed, these are single-valued, so there is no index column in this file. The parameter names are written in the first row and the values are in the second row.

#### 1.4 Solve the model

All interaction with Switch is via the command line. When you run it, it will read the files we worked with above, construct and solve the model and save the results.

On a Mac, open Terminal.app from the Applications folder. On Windows, open an Anaconda Command Prompt (Start/Windows menu > Anaconda Prompt or Start/Windows menu > Anaconda Powershell Prompt). On Linux, open the Terminal app (either search in the Dash or press ctrl-alt-F2). Below here, we will just refer to this as a command prompt or terminal window.

In the terminal window, use the "cd" command to get to the directory where you downloaded the tutorial data. (e.g., "cd Documents\tutorial" (Windows) or "cd Documents/tutorial" (non-Windows). Then type one of the following commands (press Enter or Return at the end):

Windows:

```
cd switch_tutorial\3_zone_tiny
```

Mac/Linux:

```
cd switch_tutorial/3_zone_tiny
```

Then type this command:

```
switch --help
```

You should see a list of command options (e.g., “switch solve”) and a prompt to try one of those with “--help” for more information. Try this:

```
switch solve --help
```

You will see all the options you can use with Switch or with any of the modules included in your current model. There are commands to add or remove modules relative to what is listed in `modules.txt`, or to change the amount of information shown while running, specify a solver, change input and output location, save temporary files, etc. We will solve with the “--verbose” and “--sorted-output” options. Run this command:

```
switch solve --verbose --sorted-output
```

You should see a banner, a list of arguments and modules in use, and some info on the solution status. If all went well, results have now been written to the `outputs_dir`, which is “outputs” by default.

### 1.5 View outputs

Switch back to Atom and open the “outputs” folder. You should see about 20 output files, mostly standard `.csv` files. The files with mixed capital and lower-case names are values of all the decision variables computed by Switch. They are in the same format as the inputs: index columns then values. They are all generated by the `switch_model.reporting` module.

Open **BuildGen.csv**. You will see two columns representing the indexing set (`GEN_BLD_YRS`) and one column showing the values selected by Switch. The biggest construction plans are adding 11 MW of C-Coal\_IGCC (IGCC coal in the Central zone) in 2020, 4 MW of C-Wind-1 in 2030, and 5 MW of S-NG\_CC (combined cycle gas in the South zone) in 2000 (specified in `gen_build_predetermined.csv`). Note that this file only shows additions, not retirements or cumulative capacity.

In **BuildTx.csv**, you will see a plan to add 5.8 MW of transfer capability on the C-S corridor and 2.5 MW on the N-C corridor, both at the start of the study in 2020. These are in addition to any pre-determined capacity additions. (Note that existing capacity is not included in the BuildTx variable; it is added separately, as we’ll see later.)

Feel free to browse the other variable outputs:

- **BuildLocalTD** shows the amount of intrazonal T&D capacity added in each zone during each period (generally enough to meet peak load net of distributed generation).
- **BuildMinGen** is a binary variable used to decide whether to leave a project unbuilt or build at least the minimum project size.
- **DispatchBaseloadByPeriod** shows the output level selected for baseload projects, which is assumed to be constant for a whole period.
- **DispatchTx** shows transfers along each transmission corridor during each timepoint.
- **GenFuelUseRate** shows the amount of each type of fuel (in MMBtu) used by each project during each timepoint.
- **WithdrawFromCentralGrid** shows the load in each zone during each timepoint, net of distributed resources.

You can also solve with a “`--save-expressions all`” flag to save named intermediate values that Switch calculates and uses. Those are all Pyomo Expressions that are based on the decision variables and your input data.

These outputs are very disaggregated. To get higher-level results, you can load them into data analysis software, possibly cross-referencing with input data. However, Switch also provides some additional, higher-level outputs. Some of these (`total_cost.txt` and `cost_components.csv`) are generated by `switch_model.reporting`. The rest are generated by the individual modules used for your model.

Take a look at **system\_cost.txt**. This shows that the total discounted cost for the current plan is \$ 126,750,492. This is the net present value (NPV) of all capital recovery, fixed and variable costs over the course of the study. Note that although we only include a small number of sample days, they are weighted appropriately to calculate operating costs for the whole period.

The **cost\_components.csv** file breaks this down into fixed costs for generators, intra-zonal T&D and inter-zonal transmission, and variable costs for O&M and fuel.

**gen\_cap.csv** shows cumulative installed capacity, accounting for additions and retirements, for each generation project during each investment period. It also shows some characteristics of each project, which can be used to aggregate to a higher level (e.g., generation technology, load zone, capital and O&M costs).

Other outputs include

- **costs\_itemized.csv** shows the same elements as `cost_components.csv`, broken down by study period.
- **dispatch\_annual\_summary.csv** shows total energy production, variable costs and emissions, grouped by period, technology and primary energy source and **dispatch\_zonal\_annual\_summary.csv** shows the same data broken down by load zone.



- **dispatch.csv** shows energy production, variable costs and emissions for each generation project, fuel and timepoint, weighted to reflect that timepoint's share in a typical year.
- **dispatch-wide.csv** shows power production by each generation project during each timepoint, arranged with one column per project and one row per timepoint.
- **electricity\_cost.csv** shows total cost in each period on an NPV and real basis, as well as electricity consumption and cost per MWh delivered.
- **load\_balance.csv** shows all injections and withdrawals of power at the transmission node of each zone during each timepoint. This includes total utility-scale generation, net imports, loads (net of distributed resources), and any user-specified sources or sinks.

## 1.6 Reconfigure model

Now we will change some inputs and re-run the model. We will raise the capital cost of the C-Coal\_IGCC project and see how that affects the plan.

In Atom, close any tabs that you are no longer using. Then right-click on the "inputs" folder and choose "Duplicate" (ctrl-click or two-fingered click on a Mac). Call the duplicate directory "inputs\_new".

Open `gen_build_costs.csv` inside `inputs_new`. Find the C-Coal\_IGCC project (rows 38 and 39). Type 6000000 in the `gen_overnight_cost` column for both rows to change the capital cost for 2020 and 2030 vintages from \$2,983,440/MW to \$6,000,000/MW. The new value will be stored when you press Enter or Return, but `tablr` doesn't update the display until you move off the cell. Save the file.

We're now ready to solve the model again, but we don't want to have to specify "--verbose" and "--sorted-output" every time on the command line. So instead, we'll put them in an option file that will be read every time Switch runs.

Click on the `3_zone_tiny` folder in Atom. Then choose File > New File. Enter the following lines into the window (`--stream-solver` displays messages from the solver in addition to Switch):

```
--verbose
--stream-solver
--sorted-output
```

Save the file as "options.txt" in the "3\_zone\_tiny" folder. Every time you run `switch` inside that folder, it will read `options.txt` and apply all the flags it finds there before applying the flags you specify on the command line. You can also put blank lines in `options.txt` or create comments by starting a line with "#" (the same applies for `modules.txt` and other control files). You can also put multiple options on the same line.

Now, go back to the command prompt where you solved the model in step 1.4 and run this command:

```
switch solve --inputs-dir inputs_new --outputs-dir outputs_new
```

This will solve the new version of your model and store the results in the `outputs_new` directory.

Now switch back to Atom and open the versions of `BuildGen.csv` in the `outputs` and `outputs_new` folder. You should find that Switch has abandoned the plan to build C-Coal\_IGCC in 2020 (0.0 instead of 11.12 MW before) and is now planning to build 11.4 MW of N-Coal\_IGCC (formerly 0.0).

Also open the two versions of `BuildTx.csv`. You will find that Switch has switched from building 2.5 MW of capacity on the N-C corridor to 8.4 MW, to carry the production from the N-Coal\_IGCC project to the Central load zone.

On a Mac, you can get a detailed comparison of outputs (or inputs) between two models by using `diff outputs outputs_new` or `diff outputs/BuildGen.csv outputs_new/BuildGen.csv` on the command line. You can also get a graphical comparison by using `opendiff` instead of `diff`.

## 1.7 Switch module structure

Suppose you want to check how much pre-existing transmission has been specified in a model, but don't know where it is loaded from. You may also want to check on how it is defined and incorporated into Switch. Here I will show how to do that.

In Atom, choose `File > Open...` (Mac) or `File > Open Folder...` (Windows) and then select the "switch" directory that you created with git during the pre-tutorial setup (e.g., `Documents > tutorial > switch`). This will open a new window like the one shown below.



If you have not made a local copy of the Switch source code, you can also browse it by going to <http://switch-model.org> and then navigating to the Switch repository.

If you browse through the “examples” directory, you will find a variety of examples that illustrate features of Switch. You can navigate to these on the command line (with “cd”) and use “switch solve” to solve each of them. However, for this tutorial, we will focus only on the “switch\_model” directory, where all the Switch source code resides.

In Atom, navigate to `switch_model > transmission > transport > build.py`.

As noted above, all Switch code is in the “switch\_model” directory. All transmission-related code is in the “transmission” subdirectory, and code for the commonly used transport model is in the “transport” subdirectory. Within that are several files ending with “.py”. Each of these is a Python module. The construction-related code is in “build.py”.

In Python, `switch_model` is a package. It is available to all Python programs after you install it with `conda` or `pip`. Then `switch_model.transmission.transport.build` is Python’s name for the particular module we are looking at. Every subdirectory in a Python package also has an `__init__.py` file, which contains the code that will be loaded if you access the corresponding module on its own. e.g., if you load `switch_model.transmission.transport`, Python will read the “`__init__.py`” file in this directory.

Browse through `build.py`.

When Switch runs, it loads each of the modules specified in `modules.txt` (including this one). Then it calls functions defined in these modules as needed. If a particular function is missing, Switch skips it.

The next function that Switch loads is called **`define_components()`**. This gives every module a chance to modify the shared optimization model, to define new decisions, constraints, power sources, costs, etc. Most of the `transport.build` model contains the definition for this function. This starts with a long “docstring” (in triple-quotes) describing all the components it defines. Then there are a number of lines that define Variables, Constraints, etc. These are standard objects from the Pyomo modeling library. Switch calls the `define_components()` function with a copy of the model (called “`mod`” in this case), and all the code in this function adds elements onto this model. Later, Switch will load data into the model and solve it. Once you learn the names of the components defined by the various modules and get the hang of defining Pyomo components, you will have unlimited power to extend or alter the power system model with your own modules, which will be integrated just like the standard Switch modules.

Near the bottom of the file, you will find a function called **`load_inputs()`**. This function tells Switch where to find the data for the parameters and indexes defined in this module. Switch calls this after the model has been constructed, when it is ready to load data in. Switch passes this function the shared model (“`mod`”), an object containing data to be loaded into the model

("switch\_data"), and a pointer to the inputs directory. Most data is read in using a `load_aug()` method on `switch_data`.

Finally, you will find a **`post_solve()`** function, which is called after the model is solved, and is usually used to save results.

Modules can define additional standard functions with more specialized uses. When Switch first starts up, it calls a function called "define\_arguments()" in each module (if present). This function can add command-line arguments that will be used by that module. (For examples, search for "define\_arguments" in all the source code, using Find > Find in Project.) Most standard Switch modules don't use command-line arguments, but you will often find them useful for your own modules. Other standard functions include `define_dynamic_lists()`, `define_dynamic_components` and `pre_solve()`. These are documented in section 3.3 of the supplementary material at <https://ars.els-cdn.com/content/image/1-s2.0-S2352711018301547-mmc1.pdf> (also accessible from our website). You can also find examples of how to use these functions by searching the Switch source code.

Now, let's look for information on pre-existing transmission capacity. Starting from the top of `build.py`, scroll down until you get to the component documentation. After a while, you should find a description of "existing\_trans\_cap", which is what we are looking for.

Now use Find > Find in Buffer to search repeatedly for "existing\_trans\_cap".

First, you will find it defined as a Param. The first argument(s) to the `Param()` function are the names of indexing sets. In this case, `mod.TRANSMISSION_LINES` means there will be one value of `existing_trans_cap` for each member of the `TRANSMISSION_LINES` set, i.e., `existing_trans_cap` will have a value for each transmission line. Also note that there is no "default" or "rule" argument here, so Switch cannot automatically generate a value for `existing_trans_cap`. This means the value must come from a data file. Looking further along, we see this is also reflected in the call to `min_data_check`, which will report an error if you have not provided a value for `existing_trans_cap`.

Continuing the search, you will find a definition for an Expression called `TxCapacityNameplate`. Within this, there is a rule that says that `TxCapacityNameplate` for corridor tx in a particular period should be calculated as the sum of capacity constructed in that period and all earlier periods of the study, plus `existing_trans_cap` for that corridor.

If you continue the search further down, you will find documentation in `load_inputs()` saying that `existing_trans_cap` should be read from the `transmission_lines.csv` file. Below that, you will find the code that actually reads it in, along with a number of other parameters.

If you wanted to build a whole model from scratch instead of mimicking an existing input directory, you could follow these steps:

- Browse through the `switch_model` package to find the modules you want to include.

- Read the `load_data()` function at the bottom of each module you plan to use, and identify the parameters and sets that will be read, and the files they will be read from
- Search for documentation and definitions for each parameter or set in the `define_components()` function, to see their purpose, units and how they are used in the model (you can also use Find > Find in Project to see references how other modules use each component).

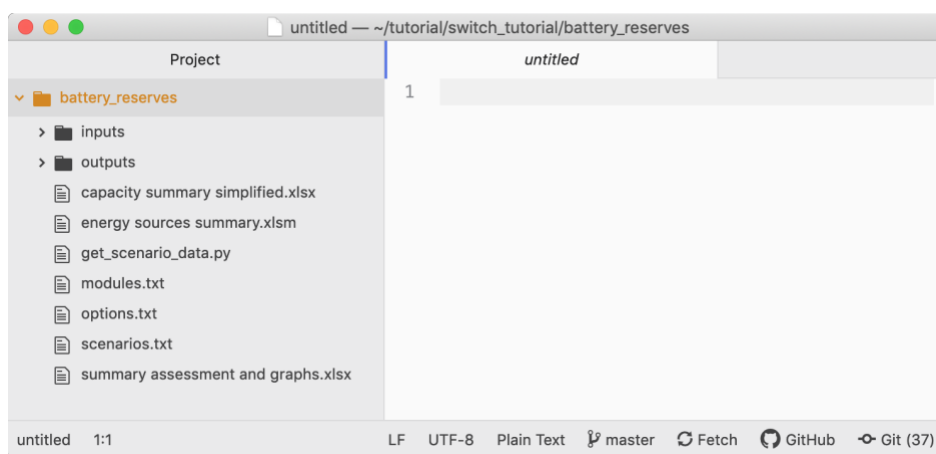
We are currently working on streamlining the source code and documentation, e.g., moving description, definition and file name together in `define_components()` and creating a website with descriptions of the columns in each table and hyperlinks to the definitions.

## 2 Comparing scenarios with Switch

In this section, we will use Switch's scenario-solving tool to define and solve several different scenarios with different inputs and command line arguments. This is a slightly simplified version of the study described in the Switch 2.0 paper (<https://doi.org/10.1016/j.softx.2019.100251>).

### 2.1 Defining and solving scenarios

In Atom, use File > Open... or File > Open Folder... to open the `battery_reserves` folder inside the `switch_tutorial` folder that you created during the pre-tutorial setup. This should create a window similar to the one below.



This folder contains inputs for a much larger model than we used before, and definitions for multiple versions of this model. The `battery_reserves` folder defines a study, and several different inputs directories and command line settings are used to define different scenarios within that study. The goal of this particular study is to estimate the value of providing operating reserves from batteries that are also used for bulk energy storage (multi-hour load shifting), instead of using only dedicated batteries for each purpose. We will also consider the cost savings available from using demand response to provide operating reserves.

Take a look at `modules.txt`. For this model, we are now using advanced versions of several either-or modules. The `energy_sources.fuel_costs.markets` module defines a supply curve

for each fuel instead of the simple linear costs used in `energy_sources.fuel_costs.simple`. The `generators.core.commit.operate` and `generators.core.commit.fuel_use` modules implement detailed unit-commitment behavior, including part-load inefficiencies, instead of the dispatch-only approach in `generators.core.no_commit`.

This model also uses a number of optional modules:

- `generators.core.gen_discrete_build` and `generators.core.commit.discrete` force projects to be built and committed in discrete chunks corresponding to a complete generating unit (a single project may represent multiple identical generating units)
- `generators.extensions.storage` allows generation projects to be identified as storage, with two-way flow of power, finite storage capacity and round-trip losses
- `balancing.operating_reserves.areas` and `balancing.operating_reserves.spinning_reserves_advanced` define spinning reserve requirements and sources. These are the main elements we will be adjusting in this study.
- a collection of modules from the `hawaii` subpackage provide either experimental new behavior (fuel market expansion via an LNG terminal) or Hawaii-specific rules and behavior.

The `spinning_reserves_advanced` module defines two classes of operating reserves: contingency reserve are used to compensate if a large power plant trips offline and regulating reserves are used for routine balancing of the system on a sub-market timescale. Generally, contingency reserves are easier to provide than regulating reserves, because they are called upon more rarely and require less communication (possibly just observation of system frequency).

When using this module, users specify named reserve products that can be provided by each generation project (by default they all provide a single product called “spinning”), and they specify which named product can be used to provide contingency or regulating reserves. Users also specify which rule to use to set targets for each class of reserves.

Look in `options.txt`. There you will see the following settings in effect:

```
--contingency-reserve-type contingency
--regulating-reserve-type regulation
--spinning-requirement-rule Hawaii
--unit-contingency
```

These specify that contingency reserves should be provided via a product called “contingency”, and regulating reserves should be provided via a product called “regulation”. They also specify that we will set operating reserve targets using a Hawaii-specific reserve rule (defined in `spinning_reserves_advanced`) and will treat outages of individual generating units as contingencies. You can see explanations of these arguments by running “`switch solve --help`” from a command line in the `battery_reserves` directory, or by looking

at the `spinning_reserves_advanced` source code or the Supplementary Material from the Switch 2.0 paper.

In `options.txt`, you will also see these settings in effect:

```
--demand-response-reserve-types regulation contingency  
--ev-reserve-types regulation contingency
```

These tell the `hawaii.demand_response_simple` and `hawaii.ev` modules that they should provide both the “regulation” and “contingency” products from their spare capacity. Demand response and EVs can provide these reserves by shifting quickly toward the minimum or maximum allowed power level.

Now look in the “inputs” directory. Here you will find three different sets of input data. These differ only in the rules about which generation projects can provide which spinning reserve products, as shown in `generation_projects_reserve_capability.csv` file. Open **`inputs/none/generation_projects_reserve_capability.csv`**. Here, all the utility-scale plants, including wind and solar, can provide the “regulation” and “contingency” reserve products, subject to the available generating capacity. There is also a dedicated contingency reserve battery (`Oahu_Battery_Conting`) that can provide only contingency reserves and a regulating reserve battery (`Oahu_Battery_Reg`) that can provide both contingency and regulating reserves. However, the bulk-storage batteries (`Oahu_Battery_4` and `Oahu_Battery_6`) are not designated as providing any reserves. (You can see more details about each of these batteries in `generation_projects_info.csv`. The reserve batteries are modeled as pure power, with no storage energy, while the bulk-storage batteries are modeled with a fixed 4- or 6-hour storage capacity.)

The specifications are the same in the **`inputs/contingency`** directory, except that the 4- and 6-hour batteries (`Oahu_Battery_4` and `Oahu_Battery_6`) can now provide “contingency” reserves, but not “regulation.” Finally, in **`inputs/regulation`**, the bulk-storage batteries can provide both “contingency” and “regulation” products.

Also note that `options.txt` specifies “`--inputs-dir inputs/regulation`”. At this point, we could go to a command prompt and run “`switch solve`” and it would solve a model where bulk-storage batteries, demand response and EVs all provide contingency and regulating reserves. However, we want to consider other scenarios, where these resources provide one or none of these services. Each of these scenarios requires different settings for various flags: `--inputs-dir`, `--demand-response-reserve-types` and `--ev-reserve-types`.

We could solve all these scenarios by running “`switch solve`” with different flags, e.g. “`switch solve --inputs-dir inputs/none --demand-response-reserve-types none --ev-reserve-types none`”. Or we could get the same effect by putting these settings in `options.txt` and then just running “`switch solve`”. However, this requires a lot of manual activity and

doesn't maintain a good record of the scenarios that were run. So instead we will use Switch's scenario-solving system.

Open `scenarios.txt` in Atom. If the lines are wrapping, use View > Toggle Soft Wrap so that you only see 5 lines that extend beyond the edge of the screen. The `scenarios.txt` file is used to define all the scenarios you want to run in your study. There is one line per scenario. Each line must include `--scenario-name` and then the name to give to this scenario. The rest of the line contains any command-line flags that you want to use for your scenario. These will be applied after flags read from `options.txt` (so they will override those), but before any additional flags you specify on the command line.

Here we have 5 scenarios. Here are the details for each one (you can ignore the `"_long"` tags):

- **battery\_bulk\_long:** batteries provide bulk storage and no reserves, and demand response provides no load shifting or reserves. This is implemented by setting these flags: `--inputs-dir inputs/none --demand-response-reserve-types none --ev-reserve-types none`. Additional flags set the time-shiftable portion of load to 0% and force EVs to charge on a business-as-usual schedule. We also set `--outputs-dir outputs/battery_bulk`, so we can keep outputs for this scenario separate from other scenarios.
- **battery\_bulk\_and\_conting\_long:** this is the same as the previous scenario, except now the bulk-storage batteries can provide contingency reserves (a relatively easy service). This uses the same settings as the `battery_bulk` scenario, but specifies `--inputs-dir inputs/contingency`.
- **battery\_bulk\_and\_reg\_long:** this is the same as the previous scenario, except bulk-storage batteries can also provide regulating reserves. This uses the same settings as before, but with `--inputs-dir inputs/regulation`.
- **dr\_bulk\_long:** this is the same as the `battery_bulk_and_reg` scenario, but now 10% of electricity demand and all EVs can provide bulk load-shifting, i.e., they can be rescheduled from one hour to another to reduce costs. However, these still cannot provide reserves. This is implemented by using the same settings as `battery_bulk_and_reg`, but with these flags: `--demand-response-share 0.1 --ev-timing optimal`.
- **dr\_bulk\_and\_reserves\_long:** this is the best-case scenario (and actually the same as the default model), where multi-hour batteries and demand response can both provide bulk load shifting and contingency and regulating reserves. This uses the same flags as `dr_bulk` but with `--demand-response-reserve-types regulation contingency --ev-reserve-types regulation contingency`.

This model is currently setup to solve with the gurobi solver. If you haven't installed gurobi, open `options.txt` in Atom. Then put a `"#"` sign in front of `--solver gurobi` to turn that line into a comment. If you have installed cplex, you can uncomment the two lines above to use that. Otherwise, Switch will automatically use glpk, which you installed earlier. After editing `options.txt`, choose File > Save.



Now we will run these scenarios.

Open a Terminal window or Anaconda Command Prompt and use “cd” to navigate to the “battery\_reserves” directory. (e.g., if you are currently in the 3\_zone\_tiny directory, you can use “cd ..” to go up one level, then “cd battery\_reserves”. Once you are in the battery\_reserves directory, run this command:

```
switch solve-scenarios
```

This tells Switch to solve all the scenarios you have defined, one after the other.

Often you will have many scenarios to run, so the solve-scenarios command allows solving multiple scenarios in parallel. To do that, try repeating the previous step 1-3 more times (depending how many cores you have). i.e., open some more command prompts, navigate to the battery\_reserves directory, and run “switch solve-scenarios” in each one. You should see messages in each window identifying the scenario that is currently being run. Switch will automatically run all the scenarios and avoid running the same scenario twice.

You can use this same technique to solve multiple scenarios on a supercomputing cluster—switch solve-scenarios can be run on separate machines, as long as they share the same model directory. If you install openmpi (“conda install openmpi”), you can also launch multiple instances of switch with a single command: “mpirun switch solve-scenarios”. Note that output from the different instances will be mixed; you may want to use the --logs-dir option for each scenario to keep the logs distinct.

These models are large and will each take 20-40 minutes to solve. You can wait for them to finish and save results in the “outputs” directory. Or, to move on with the tutorial more quickly, you can interrupt them with ctrl-C and use outputs that have already been saved there.

## 2.2 Analyzing results

The results from all five scenarios are saved in subdirectories of battery\_reserves/outputs, with names matching each scenario. These files are standard .csv files, so you can analyze them with virtually any tool. For this tutorial, to avoid turning this into a data-analysis class, I will use Microsoft Excel and minimal automation. You may prefer other tools or a mix of tools (I recommend Python with matplotlib or R). If you have Excel installed on your own computer, you can follow along with that, or otherwise you can watch on the screen.

### 2.2.1 Compare costs across scenarios

Here we will compare total costs per customer across all 5 scenarios, to see how valuable the various interventions are.

Switch to Excel and create a new workbook with entries like this starting in cell A1:

scenario	total cost
battery bulk	
battery bulk and conting	
battery bulk and reg	
dr bulk	
dr bulk and reserves	

Switch back to Atom. Open the “outputs” folder, and inside that open the “battery bulk” folder. Scroll down and click or double-click on “total\_cost.txt”. This file has the total, discounted cost for the scenario. Select the text in the window and copy it (ctrl-C or command-C). Switch back to Excel and paste into the first row below total cost.

Repeat the process above for all 5 scenarios. You will end up with a table like this:

scenario	total cost
battery bulk	1.7346E+10
battery bulk and conting	1.7311E+10
battery bulk and reg	1.713E+10
dr bulk	1.6311E+10
dr bulk and reserves	1.6297E+10

In the next column to the right, create a new label, “cost per customer”. In the first cell below that, fill in the formula “=B2/460000” (Oahu has 460,000 customers). Then select from there to the bottom of the column, then choose Home > Editing > Fill > Down (or just press ctrl-d).

Now, label the next column over “savings per customer”. In cell D2 below that, fill in the formula “=C5-C2”. This will subtract the adjacent cell from C2 and report the result. Fill down to the bottom of the table. Now you will have a table like the one below:

scenario	total cost	cost per customer	savings per customer
battery bulk	1.7346E+10	37708.3516	0
battery bulk and conting	1.7311E+10	37633.2376	75.1139321
battery bulk and reg	1.713E+10	37238.9421	469.409489
dr bulk	1.6311E+10	35458.1434	2250.2081
dr bulk and reserves	1.6297E+10	35428.5362	2279.8154

Inspecting this, we conclude that the savings are not large when bulk-storage batteries also provide contingency reserves (\$75 over 30 years). They are bigger, but not huge, when the batteries provide regulating reserves (\$469 over 30 years). Turning to demand response, if 10% of demand can be shifted freely from hour to hour, that would have a lot of value—\$2,250 per

customer over 30 years. That may be enough to justify investments in price-responsive controllers. However, if demand response could also provide contingency and regulating reserves it wouldn't add much more value—only another \$29 per customer, which is probably not enough to justify investing in the extra communication and control system. (Digging deeper, you would find that bulk storage batteries naturally provide enough reserve capacity that adding more has little value.)

On Monday, I added a script to the “battery\_reserves” directory called “gather\_total\_costs.py”. This shows an alternative way to compare these costs, automating the steps above. If you don't have that script, you can follow the steps at the start of section 2.2.3 to get it. Once you have this script, you can run it by going to a command prompt in the battery\_reserves directory, and then running “python gather\_total\_costs.py”.

### 2.2.2 Inspect hourly behavior in each scenario

If you have Excel installed, use it to open the “energy\_sources\_summary.xlsm” file from the battery\_reserves folder. If you enable macros, it will make it a little easier to swap between files, but you can also disable macros if you prefer.

On the “**data**” worksheet, you will find a copy of data from “energy\_sources\_<scenario name>.csv”, saved in the outputs directory by switch\_model.hawaii.save\_results (in the future, you could create your own custom modules to create custom output). This table is pretty messy, because it has been revised over the years to meet different needs. The first few columns identify the load zone, study periods and timepoints. For every timepoint, it shows total production from various primary energy sources (columns D-M), then production from various technologies (columns N-W), then some information on curtailment, then total centralized and distributed production, then all types of loads (columns AD–AG), then spinning reserve supply and demand, then marginal cost (not calculated in these scenarios), then a label for peak vs. typical days.

This is a lot of information, and many of the columns are generated automatically, so they may appear and disappear from one model to another. The number of timepoints may also change from one study to another. I deal with these variations by creating a second worksheet, “**filtered**”, which selects particular columns by name from the data worksheet (or replaces missing values with 0), and then also generates new columns based on them (starting at column AG). This is used to create the columns and rows needed for graphing. This is useful for interactive examination of model results but is a little brittle. If you consistently need to produce one particular type of graph, you may prefer just to write scripts for that purpose in Python or R.

On the “**graphs**” worksheet, you will see graphs of the hourly behavior of the power system, derived from the “filtered” worksheet. Note that for these sample days in 2045, the power system is dominated by solar (yellow), with a little wind (blue) and biofuel (green).

Switch to the “filtered” page. Click on the header for the “period” column and choose a different period, e.g., 2020 (this is just Excel’s standard auto-filter tool). Switch back to the “graphs” page. Now you will see that the least-cost choice for Oahu for 2020 is a lot less solar and still uses a fair amount of low-sulfur fuel oil (LSFO). Now switch back to the “filtered” page and switch back to 2045, then go to the graphs page again.

When you first load it, this file shows the hourly behavior of the “dr bulk and reserves” power system. Note that this system has a lot of reserves available at all times (peaking above 5 GW total production and reserves each day) and also never uses more than about 500 MW from batteries.

Click on the “Choose Data File” button. This causes the data table on the “data” page to refresh and prompt for a new file. (If you have not enabled macros, just right click anywhere in the data table on the “data” page and choose “Refresh” instead.) When prompted, navigate to battery\_reserves > outputs > battery\_bulk > energy\_sources\_battery\_bulk\_long.csv”. Now the available reserves are much lower (closer to the required amount), and battery use is sometimes above 1000 MW.

Feel free to load other energy\_sources files or switch to different years.

### 2.2.3 Compare capacity of different generation technologies across scenarios (advanced)

Switch back to the terminal window or command prompt you have open on the “reserve\_study” directory.

If you installed the tutorial data before 1:30 pm on Monday, please pull the later version, which has a script to aggregate capacity investment data into a form that is easy to plot in Excel. To do this, make sure you have an Internet connection, then run this command:

```
git pull
```

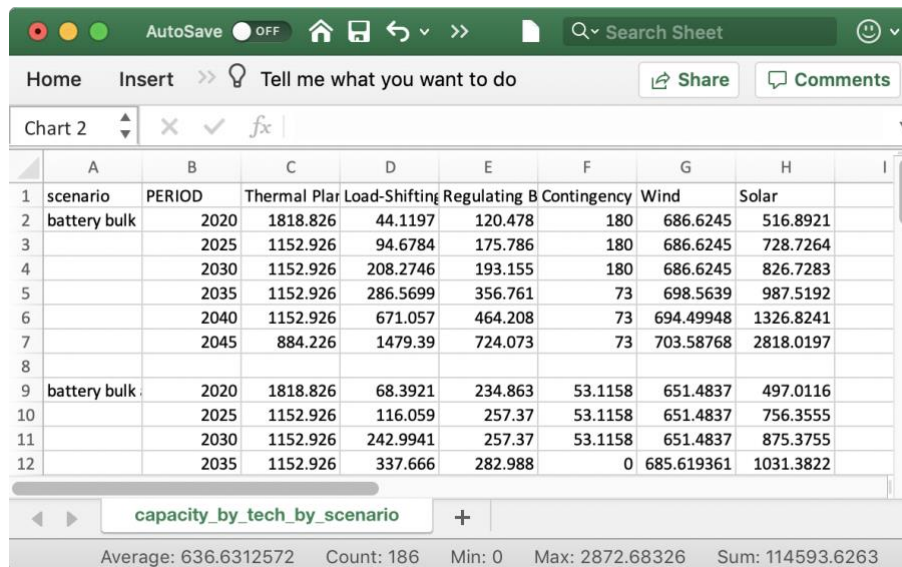
If you are unable to get the latest version of the tutorial data, you won’t be able to run the next step, but you can follow as I show it on screen.

At the command prompt, next run this command.

```
python aggregate_capacity.py
```

You should see a message that capacity\_by\_tech\_by\_scenario.csv was saved in the outputs directory. This script uses the “Pandas” Python package to collect data from the gen\_cap.csv files in all the outputs directories, aggregate it into major categories, create columns for each category, then reformat the data for easy plotting in Excel. If you are curious about how to do this, please feel free to look at the script in Atom. (If you have installed the Hydrogen package, you can even run the script from there.)

Switch to Microsoft Excel. Use File > Open and navigate to the outputs directory and open `capacity_by_tech_by_scenario.csv`. (Alternatively, you could right-click on the file in Atom, choose “Reveal in Finder” (Mac) or “Show in Explorer” (Windows), then double-click on the file to open it in Excel.) You should see a file like this:



	A	B	C	D	E	F	G	H	I
1	scenario	PERIOD	Thermal Plar	Load-Shifting	Regulating B	Contingency	Wind	Solar	
2	battery bulk	2020	1818.826	44.1197	120.478	180	686.6245	516.8921	
3		2025	1152.926	94.6784	175.786	180	686.6245	728.7264	
4		2030	1152.926	208.2746	193.155	180	686.6245	826.7283	
5		2035	1152.926	286.5699	356.761	73	698.5639	987.5192	
6		2040	1152.926	671.057	464.208	73	694.49948	1326.8241	
7		2045	884.226	1479.39	724.073	73	703.58768	2818.0197	
8									
9	battery bulk	2020	1818.826	68.3921	234.863	53.1158	651.4837	497.0116	
10		2025	1152.926	116.059	257.37	53.1158	651.4837	756.3555	
11		2030	1152.926	242.9941	257.37	53.1158	651.4837	875.3755	
12		2035	1152.926	337.666	282.988	0	685.619361	1031.3822	

Summary statistics: Average: 636.6312572, Count: 186, Min: 0, Max: 2872.68326, Sum: 114593.6263

Next, select all the data cells and headers (C1:H35) but not the row labels. Then go to the Insert tab, click the dropdown arrow next to the column chart icon, and choose a 2D stacked column chart. You should then get a chart, and the Chart Design tab will activate.

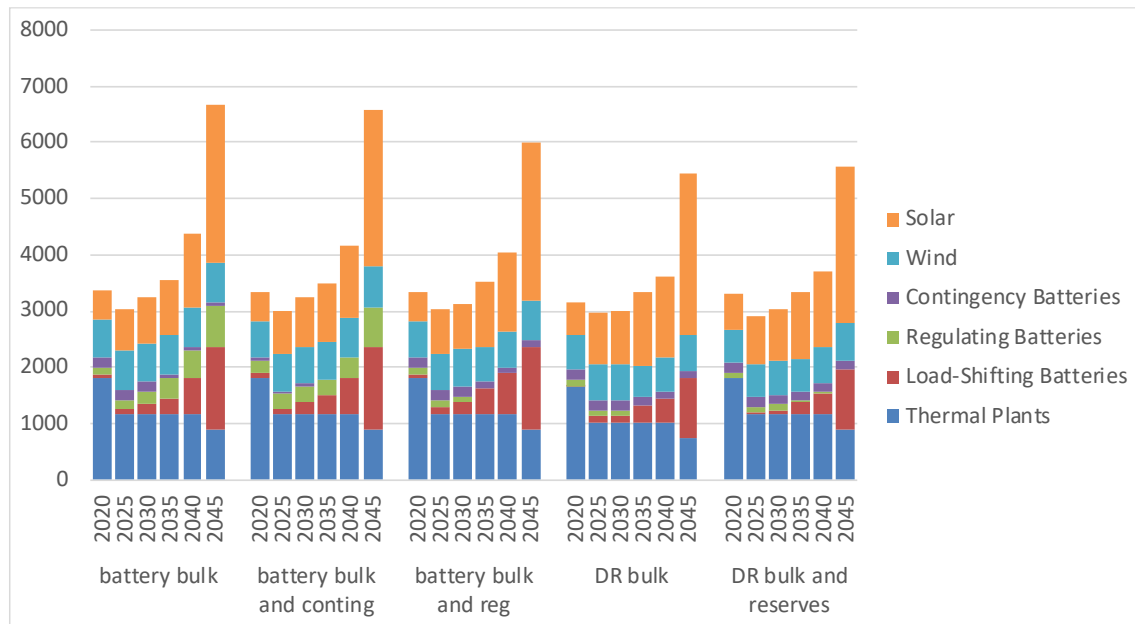
Click “Select Data” on the Chart Design tab. Click in the “Horizontal (Category) axis labels” box, then select all the scenario names and dates. The box should say “=capacity\_by\_tech\_by\_scenario!\$A\$2:\$B\$35”. Click OK.

Make the chart bigger and format as desired. I recommend these steps:

- While you have the chart selected, go to the Chart Design tab and choose Add Chart Element > Legend > Right.
- While the whole chart is selected, choose a larger font size on the Home tab, e.g. 14 pt.
- Double-click one of the series to open the “Format Data Series” pane, then click on the graph icon, then under “Series Options” choose Gap width: 20%.
- Double-click on the bottom axis to open the “Format Axis” pane, then click on the graph icon, then under “Labels” choose Specify interval unit: 1
- Click on the chart title and then press the delete key to delete it.

Now the graph should look like the one below. We can inspect it to see the most significant changes between scenarios. These seem to be that contingency batteries mostly drop out when going from “battery bulk” to “battery bulk and conting”. Then regulating batteries drop

out when going to “battery bulk and reg” (partly replaced by contingency, partly just eliminated). Finally, bulk battery investments are reduced when demand response is available.



### 3 Customizing Switch

#### 3.1 Writing a custom module (cogen example)

In this section, we will see how to write examine a custom module that extends the `3_zone_tiny` model. We will represent a hypothetical cogen technology that can be added to any thermal power plant to produce electricity from waste heat. This technology will have the following properties:

- waste heat is converted to electricity load reductions at a rate of 20 MMBtu per MWh
- carrying cost is \$150,000 per MW of output capability per year, including capital recovery and O&M
- once built, capacity cannot be retired (this makes availability and cost calculations easier)
- if not needed, cogen load reductions can be discarded

To implement this, we will need two new decision variables: how much cogen capacity to add at each thermal plant during each period and how much power to produce from the cogen equipment during each timepoint. We will also need to add constraints so that the power production doesn't exceed the installed capacity or the available waste heat. We will also need to add the power production to the system energy balance and add the cost of the cogen equipment to the system costs. In this section, we will examine a custom module that implements these elements.

In Atom, use File > Open... or File > Open Folder... to open the 3\_zone\_tiny model.

Choose File > New. This will create a new tab. We will store the parameters for this model here. On the first row, enter the new parameter names, separated by commas, with no spaces or quotes: “cogen\_heat\_rate,cogen\_fixed\_cost”. On the second row, enter the values: “20,150000”. Then choose File > Save and save it as “cogen.csv” inside the “inputs” directory.

Double-click on “cogen.py” in the “3\_zone\_tiny” directory to open it in a new tab. We will go through it line by line.

At the top, you will find two import statements:

```
import os
from pyomo.environ import *
```

These import the standard Python “os” module (later used to manipulate paths) and the standard definitions of Pyomo objects. These are used to define Switch model components.

The main code for this module is in a function that starts with

```
def define_components(m):
```

This creates a “define\_components” function that accepts a model called “m”. Switch will call this function as it builds the model, and this function will attach the cogen components to the model.

It’s common to start by defining the model parameters (data components) that will hold values read into the model:

```
    m.cogen_heat_rate = Param()
    m.cogen_fixed_cost = Param()
```

Note that these lines (and everything else in define\_components()) is indented by 4 spaces. In Python, everything inside a control structure (e.g., function definition, “for” loop, “if” statement) must be indented consistently. This is how Python knows which code is included in that element and what isn’t. As you write code, Atom will automatically indent on the next line after you start a function, and it will keep the indents the same as you add new lines. If it doesn’t get the indents right, you can always press tab to indent or backspace to delete an indent. Once you’ve started writing a line, you can also use cmd-] or ctrl-] anywhere on the line to indent it one step further, or cmd-[ or ctrl-[ to reduce the indentation by one step.

For future reference, all sets in Switch are written in UPPERCASE, parameters (like `cogen_heat_rate`) are written in lowercase, variables and expressions are written in MixedCase and constraints are written in Mixed\_Case\_With\_Spaces.

Now we need to define a `BuildCogen` variable for each thermal plant for each study period and a `DispatchCogen` variable for each thermal plant for each timepoint. If you read the Supplementary Material for Switch, you will find that Switch has separate sets of fuel-powered generators and non-fuel-powered generators, and the set of fuel-powered generators is called `FUEL_BASED_GENS`. Fuel-powered generators are the same thing as thermal plants, so those are the ones we will focus on for cogen. You can also tell from the Supplementary Material or from `switch_model.timescales` that the set of all study periods is called `PERIODS` and the set of all timepoints is called `TIMEPOINTS`. With that in mind, we can define the variables like this:

```
m.BuildCogen = Var(  
    m.FUEL_BASED_GENS, m.PERIODS,  
    within=NonNegativeReals  
)  
m.DispatchCogen = Var(  
    m.FUEL_BASED_GENS, m.TIMEPOINTS,  
    within=NonNegativeReals  
)
```

Note that Python allows you to wrap lines and indent code any way you want inside of parentheses or brackets. The formatting in this tutorial is consistent with the Python style guide (<https://www.python.org/dev/peps/pep-0008/>).

These `Var()` definitions follow a common pattern for Pyomo objects (and therefore Switch components), such as `Param()`, `Var()`, `Constraint()` or `Expression()`. The first argument(s) are the indexing set or sets it will use. These indicate that there will be one value for each member of the indexing set or each possible combination of the indexing sets (e.g., for each fuel-based generator during each period). The two Params we defined earlier didn't have indexing sets, so they were single-valued. After the indexing sets come various named arguments, e.g., a domain to restrict values to. Both of these variables must take non-negative real values, which is very common in linear optimization models.

Now we need to calculate the amount of cogen capacity that will be online at each project site during each period. This can be done as follows:



```

def CogenCapacity_rule(m, g, p):
    capacity = sum(
        m.BuildCogen[g, p2] for p2 in m.PERIODS if p2 <= p
    )
    return capacity
m.CogenCapacity = Expression(
    m.FUEL_BASED_GENS, m.PERIODS,
    rule=CogenCapacity_rule
)

```

This follows another common pattern for Pyomo Constraints and Expressions. Looking at the second half of this code, we see that an Expression() is defined with zero or more indexing sets and each value is calculated via a rule. The rule is a normal Python function that accepts references to the model and one member of each indexing set, and returns the correct value for those. Here there will be one value for each cogen site for each period, and each value will be calculated by calling the CogenCapacity\_rule function with the model (m), generation project (g) and period (p).

Looking at the first half of this code, we see the definition for the rule function. This calculates the online capacity as the sum of all capacity built at that project during this period or prior periods. In Python, the sum() function can add up any list of items. Here, we used a “list comprehension” to generate these items. List comprehensions are a powerful tool in Python for quickly generating a lists of elements. Here, the list comprehension takes each period in turn (from the set m.PERIODS), assigns it to the variable p2, checks whether that is less than the period currently being considered (p), and if so, it retrieves the amount of cogen built at the current project (g) in in period p2 (BuildGen[g, p2]). Then the sum() function adds up all these values and the total is returned as the value for CogenCapacity[g, p].

As written so far, the DispatchCogen variable could be set to any nonnegative value. We need to restrict it so it doesn't exceed the available cogen capacity or waste heat. The first part of that is done as follows:

```

def Max_DispatchCogen_rule(m, g, t):
    test = (m.DispatchCogen[g, t] <= m.CogenCapacity[g, m.tp_period[t]])
    return test
m.Max_DispatchCogen = Constraint(
    m.FUEL_BASED_GENS, m.TIMEPOINTS,
    rule=Max_DispatchCogen_rule
)

```

In Pyomo (and Switch), Constraints are written similarly to Expressions, but they return True or False values. The solver will ensure that variables are chosen for these variables that keep the constraint rule true. This constraint applies to each cogen location during each timepoint. In the second line, we needed to cross-reference the study period that timepoint t belongs in, in order

to lookup how much cogen capacity is online. That is done via the `tp_period[t]` parameter. You can find definitions for helpers like this in the `switch_model.timescales` code.

Now we need to limit the power output from the cogen unit based on its heat rate and the available waste heat. If you look in the source code of `switch_model.generators.core.no_commit` or the Supporting Material, you will see that the fuel used with the `no_commit` module is just `DispatchGen * gen_full_load_heat_rate` (in MMBtu). We can also convert the electrical output into MMBtu by multiplying `DispatchGen * 3.412`, which is the number of MMBtu in 1 MWh. By the First Law of Thermodynamics, the difference between these is the generator's waste heat (in MMBtu). We can also find the required heat for the cogen unit by multiplying `DispatchCogen * cogen_heat_rate`. So the following code will restrict the output from the cogen unit so that its heat consumption is not greater than the available waste heat:

```
def DispatchCogen_Available_Heat_rule(m, g, t):
    if (g, t) in m.GEN_TPS:
        # this is a timepoint when the thermal plant can run
        heat_input = m.DispatchGen[g, t] * m.gen_full_load_heat_rate[g]
        work_done = m.DispatchGen[g, t] * 3.412 # power output in MMBtu
    else:
        # thermal plant is pre-construction or retired
        heat_input = 0
        work_done = 0
    rule = (
        m.DispatchCogen[g, t] * m.cogen_heat_rate # heat used by cogen
        <= heat_input - work_done # heat available
    )
    return rule
m.DispatchCogen_Available_Heat = Constraint(
    m.FUEL_BASED_GENS, m.TIMEPOINTS,
    rule=DispatchCogen_Available_Heat_rule
)
```

Note that we have defined the cogen units as being able to produce power in all timepoints of the study, but the standard generating units can only produce power between their allowed construction years and the times when they reach maximum age and retire. Switch conserves memory by only defining decision variables for generators during the times when they can potentially be operated. Switch defines a set `GEN_TPS` that shows valid pairs of generation project `g` and timepoint `t`. In the code above, we double-check whether `t` is an allowed operating time for generator `g` by checking if they are in this set. If not, we assume waste heat is zero. The `GEN_TPS` set and many other related sets are defined in `generators.core.dispatch` and `generators.core.build`.

We are nearly done, but we need to add the output from the cogen units to the system energy balance, and we need to add the costs to the system cost calculation. Otherwise the cogen units will be useful or they will be built without regard to cost.

To add elements to the system energy balance, we first define a Pyomo expression or variable that shows the total power production from that type of element in each load zone during each timepoint. Then we append the name of the summary component to a list called “m.Zone\_Power\_Injections”. (Demand can be added to the system by appending a similar component to m.Zone\_Power\_Withdrawals.) The code below does this.

```
# calculate power output from cogen units in zone z in timepoint t
def CogenZonalOutput_rule(m, z, t):
    total_output = sum(
        m.DispatchCogen[g, t]
        for g in m.FUEL_BASED_GENS
        if m.gen_load_zone[g] == z
    )
    return total_output
m.CogenZonalOutput = Expression(
    m.LOAD_ZONES, m.TIMEPOINTS,
    rule=CogenZonalOutput_rule
)
# Add cogen output to system generation balance
m.Zone_Power_Injections.append('CogenZonalOutput')
```

This code is similar to what we have previously written. This time, one value of the expression is defined for each load zone and timepoint. The one new element is that we have used the parameter m.gen\_load\_zone[g] to lookup the load zone where generator g is located, and then check whether that matches the load zone we are interested in. (This is an inefficient way to do this, because it checks all thermal generators instead of just iterating over the ones in this load zone. However, it easy to write, so we use it in this example. In the main Switch code, you will see many cases where we generate specialized sets, e.g., all fuel-based generators in zone z, and then iterate over those to avoid this inefficiency.)

Finally, we need to add the cost of the cogen units to the total system cost, so their construction can be co-optimized with everything else. We have kept the cost structure simple here—it will just be an annual fixed cost based on the total installed capacity. Annual costs are added to Switch by defining an expression that reports the total annual cost for this element during each study period. Then the name of this expression is appended to a (slightly misnamed) list called “Cost\_Components\_Per\_Period”. Here is some code to do that for the cogen units:

```
# Calculate fixed costs for all cogen units online in period p
def CogenFixedCost_rule(m, p):
    total_capacity = sum(m.CogenCapacity[g, p] for g in m.FUEL_BASED_GENS)
    return total_capacity * m.cogen_fixed_cost
# Add fixed costs to model
m.CogenFixedCost = Expression(m.PERIODS, rule=CogenFixedCost_rule)
m.Cost_Components_Per_Period.append('CogenFixedCost')
```

This completes the definition of the cogen units. Now we also need a little code to read the `cogen_heat_rate` and `cogen_fixed_cost` parameters from `cogen.csv`. To do that, we define another function called “`load_inputs`” at the top level of the `cogen.py` module (i.e., the `def` statement should not be indented at all). Switch will call this function after the model is defined to read in values that are stored on disk. Every parameter or set in the model must receive a value either via an initialization rule, a default rule, or by being read from disk by `load_inputs`. Here is the `load_inputs` code for the cogen module:

```
def load_inputs(m, switch_data, inputs_dir):
    switch_data.load_aug(
        filename=os.path.join(inputs_dir, 'cogen.csv'),
        autoselect=True,
        param=(m.cogen_heat_rate, m.cogen_fixed_cost))
```

The `load_inputs` function receives a reference to the model (`m`), a reference to a Pyomo DataPortal object (`switch_data`) which will hold all the data before it is loaded into the model, and the name of the inputs directory (`inputs_dir`). The `switch_data.load_aug()` function is similar to Pyomo’s built-in `DataPortal.load()` function, but has been enhanced to deal with optional inputs and data files. See the first part of this tutorial for a little more information about this function.

After all that work, we’re now ready to run the model using the custom module.

First, in Atom, open `modules.txt` in the `3_zone_tiny` directory and add a new line that says “`cogen`”. This will cause your module to be loaded when Switch runs. (Alternatively, you could add “`--include-module cogen`” to the command line.)

Go to a Terminal window or Anaconda Command Prompt and navigate to the `3_zone_tiny` directory. Then run “

```
switch solve --outputs-dir outputs_cogen
```

This will run your model and save the results in the `outputs_cogen` directory.

You can now use Atom (or the “`fc`” command on Windows or “`diff`” command on other platforms) to compare results between here and the original “`outputs`” directory. You will

find that the cogen option has reduced `total_cost` from \$127 million to \$118 million. You will also find that it has been added mainly to the C-Coal\_IGCC and C-Coal\_ST plants. They also cause construction of C-Coal\_IGCC in 2020 to be reduced from 11.1 MW to 8.2 MW and cause a slight increase in N-Wind-1 (see `BuildGen.csv`). This may be because the cogen is dispatchable while the IGCC needed to run at full power at all times.

### 3.2 Debugging models

When writing your own modules, you will often run into errors, and unfortunately, they can be quite cryptic.

Syntax errors will pop up when you first run the “switch solve” command for a model that includes your module. If you want to identify these more directly, you can just run “python mymodule.py”. This causes Python to load your module and run all the top level code. Since Switch modules just define functions at the top level, nothing very interesting will happen. But Python will report any syntax errors or top-level import errors.

Once you have sorted out the syntax errors, you will likely find that your module references components that don't exist or uses indexes incorrectly, or makes other logical errors. These will be harder to pin down than syntax errors, because they generally occur somewhere deep in Pyomo code. When you run “switch solve”, Python runs Switch, which in turn loads all the modules you have specified. It then runs all the `define_components` and `load_inputs` functions to define the model and read raw data from the disk. None of your rules run at this point—they are just compiled and attached to the model components for use later. Next, Switch passes the model definition and data object to Pyomo, and Pyomo goes through each model component (Variable, Expression, Constraint, Parameter, etc.) in the order it was defined. It runs the rule function for that component if available, or reads data from the data object or calculates default values using a default rule. If this all succeeds, then Switch calls Pyomo to solve the model, and Pyomo writes the model to disk in standard form for a solver, calls the solver and loads results back into the model. Finally, Switch calls all the `post_solve` functions in your modules.

Errors often show up at the stage where Pyomo is trying to attach data to your model. Python will report an error and show the stack of functions that were called, but you will barely see any code you recognize.

Sometimes an error will occur because modules are not loaded in the right sequence. If one module reports an error that a component is not constructed, search the Switch repository to find where the component is defined (searching for “Component =” will usually find the definition). Then make sure that module comes earlier in `modules.txt`.

Often errors will occur because you supply invalid or missing data (e.g. values with invalid index keys). You can diagnose these errors by checking your data, but you may be able to help diagnose them by running “switch solve --debug”, which will launch the Python debugger at the

point where the error occurs. This is often useful for analyzing errors that interrupt execution of your model, either in your code or in the core Switch code.

See documentation on the Python debugger (<https://docs.python.org/3.7/library/pdb.html>) for commands you can use here. Often the most useful command to run is “dir()”, because that will show you the variables defined in the current operating environment. You can also use “dir(object)” to see all the methods and properties that “object” has. Then you can use “u” (up) and “d” (down) to move to a level where there are objects you recognize (e.g. the Switch model or DataPortal object). If the DataPortal object is called “switch\_data”, you can look at all the data that has been loaded via “p switch\_data.data()”. Or you can narrow it down a little with things like “p switch\_data.data().keys()” or “p switch\_data.data()['fuel\_cost']”. If the model is in an object called mod, you can also see a nicely formatted description of the model via “mod.pprint()”, or you can inspect individual components via “mod.fuel\_cost.pprint()”. This is only recommended for small models, because it generates *a lot* of output.

For errors that don’t crash your model but give unexpected results, you may need to inspect your model to understand them. There are a few ways to do that. One is to put print() statements in your code (or the Switch or Pyomo code) to show progress, intermediate values, etc. Alternatively, you can invoke the Python debugger at any point in the code by adding the line “import pdb; pdb.set\_trace()”. Then you can use the commands from the previous paragraph. Alternatively, if your model is running but giving unexpected results, you can run “switch solve --interact”. This will complete the solution process and then place you at a normal Python command prompt (not the debugger), with your model loaded into a variable called “m”. Then you can use the commands from the previous paragraph, minus the “p” prefix, to inspect the model. You can also look at values of individual components by using something like “value(fuel\_cost[‘zone1’, ‘coal’, 1234])”. The “value” call is needed to distinguish between looking at the Pyomo object and looking at its current value.

Infeasible models can also be tricky to diagnose. For these, your main options are (1) keep removing rules (usually in your own code) until it becomes feasible, then put them back until it becomes infeasible again. Or (2) direct CPLEX to return an irreducibly inconsistent set of constraints or gurobi to return a irreducibly inconsistent subsystem. These will show you a relatively small number of constraints that cannot be met simultaneously, which should help you spot the problem. The settings to get these are a little tricky. For cplex, you can use try these settings: --solver=cplex --solver-io=nl --solver-options-string="display=1 iisfind=1" --stream-solver --suffixes iis . Check the solver documentation or contact me (Matthias Fripp <mfripp@hawaii.edu>) for help with other solvers.

### 3.3 Input data workflows

You will generally want to gather your raw input data into files in logical locations (can be Excel files, .csv files, shape files, etc.). Then you will want to use scripts to convert these into Switch inputs for each study. There are two main approaches for these scripts:

1. Write a script that converts the raw inputs directly into input files for Switch model runs. If using Python, you will find the pandas library invaluable for this. The script should have adjustable settings (command line arguments or settings at the top) to control things like whether to use high/med/low fuel cost, equipment cost or load cases, and how to define study periods and sample timeseries. Then it should read the correct input files, sub-sample timeseries as specified, translate timeseries data for future periods as needed, and write all the input files we have discussed here.
2. Write three scripts:
  - a. One script converts raw inputs into standard tables and stores them in a database in standard form. It is helpful to structure your database so the tables have flags for different versions of the data, so e.g., you could have five different EV adoption schedules, each with a different “ev\_scenario” flag. You could also have different flags for different capital cost scenarios, fuel cost scenarios or load scenarios. This script should also define different time sampling patterns that you may use, and store the corresponding periods, timeseries and timepoints in tables, along with flags identifying which sample they are. As you think of new studies to perform with new data, you can expand your data-ingest script to add the additional data to your database along with the older data. You can also run only sub-parts of the data-ingest script as needed, to add new data. (Generally, the script should clear out records of a particular type before regenerating them.)
  - b. One script (or Python module) generates Switch inputs from the database, using arguments passed to it by another script. This script does all the work of creating inputs for a particular study, using filters and selectors that are passed to it (e.g., use the “2045\_dense” time sample with the low EV adoption scenario, medium EIA fuel cost forecast, medium ATB equipment cost forecast and medium load forecast). The `switch_model.hawaii.scenario_data` module is an example of this type of script.
  - c. One script is written for each study, that calls your data export script with the arguments needed to define each inputs directory that is needed for the study (it will likely call the export script multiple times, with different arguments). This script can also create a `scenarios.txt` file to define your scenarios. The `get_scenario_data.py` script in the `battery_reserves` tutorial directory is an example of this type of script.

The first approach works well for one-off projects, or projects where the requirements are clear from the start. However, it can be difficult to maintain if you have many different studies with different mixes of required data. It can also be difficult to share with colleagues, who will all need copies of all your input data. On the other hand, you don't need a database, and if the inputs are small you can share them via github, Dropbox or Google Drive.

The second approach works well if you need to run multiple heterogeneous studies from a common dataset, or you have a growing collection of inputs that you need to mix and match over time. It can also support sharing with colleagues without copying large datasets. On the

other hand, you need to learn yet another language to manage the database, design your database schema carefully, and perform regular backups (unless you're sure you can regenerate the database completely from your ingest code in a reasonably short time). You also need to manage credentials and security for people you will share with, and there are no best practices for providing database access to the general public. We are working on moving the Hawaii data warehouse from a local server to a cloud database that supports public data, to enable broader sharing of the back-end data.

One guiding rule: use saved scripts for all your data manipulation so you can replicate, document or change the process later. Don't use ad hoc database queries or create input files by hand because then you will have no automated way to redo and tweak them later (and you will always need to redo and tweak them).