Switch 2.0 Tutorial

> Before you begin the tutorial, please complete the steps described in the "Pre-Tutorial Setup" document to install Switch on a laptop computer and learn some background information.

In this tutorial, required steps are marked by a blue bar 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.

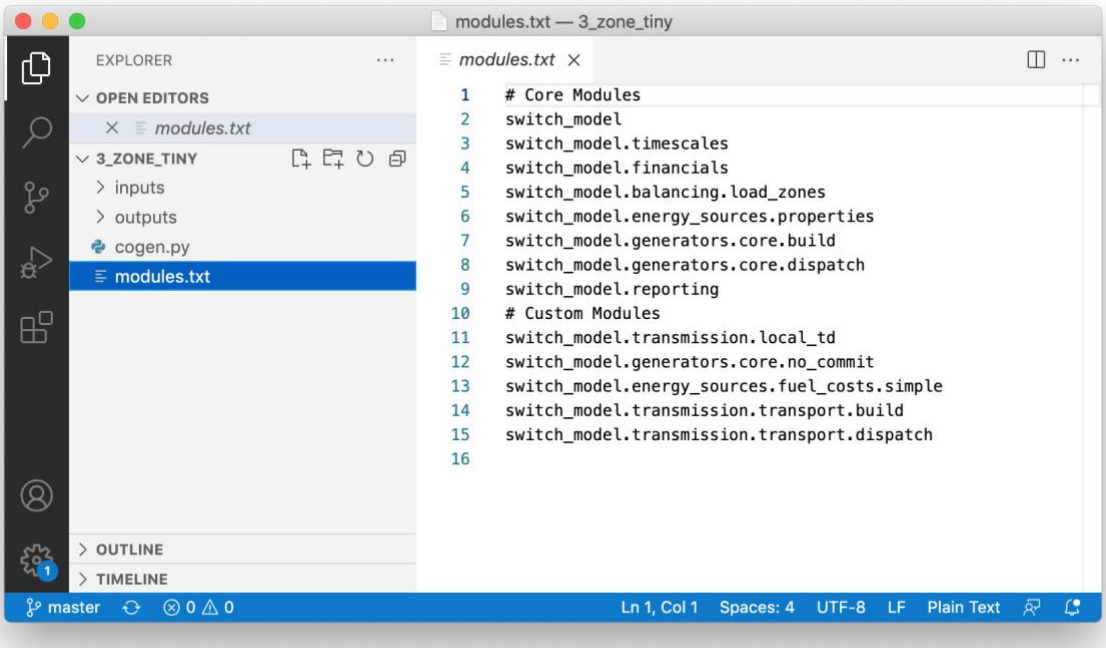
- The project website at http://switch-model.org is still pretty minimal, but it has shortcuts to the documentation listed below.
- The Switch 2.0 paper at https://doi.org/10.1016/j.softx.2019.100251 provides a concise overview of Switch 2.0 and is recommended for all users.
- Section 4 of the Supplementary 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", "switch solve --help" (in a model directory) or "switch solve-scenarios --help" 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 VS Code to find all references to a particular component. Reading the load_inputs section at the bottom of each module is one of the best ways to identify which input tables (csv files) and columns need to be defined to use 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" inside a tutorial folder on your computer. Now we will work with the simple model called "3_zone_tiny" inside this folder.

> Open VS Code, then choose File > Open Folder… and select the 3_zone_tiny folder (not the files inside, just the folder) and click OK. Close the "Getting Started" tab if it appears. Click on View > Explorer if you don't see the Explorer pane on the left. Click on "3_zone_tiny" in the Explorer pane to expand the directory list, then click "inputs", then on "modules.txt" inside the "inputs" folder. This should create a window similar to the one below.



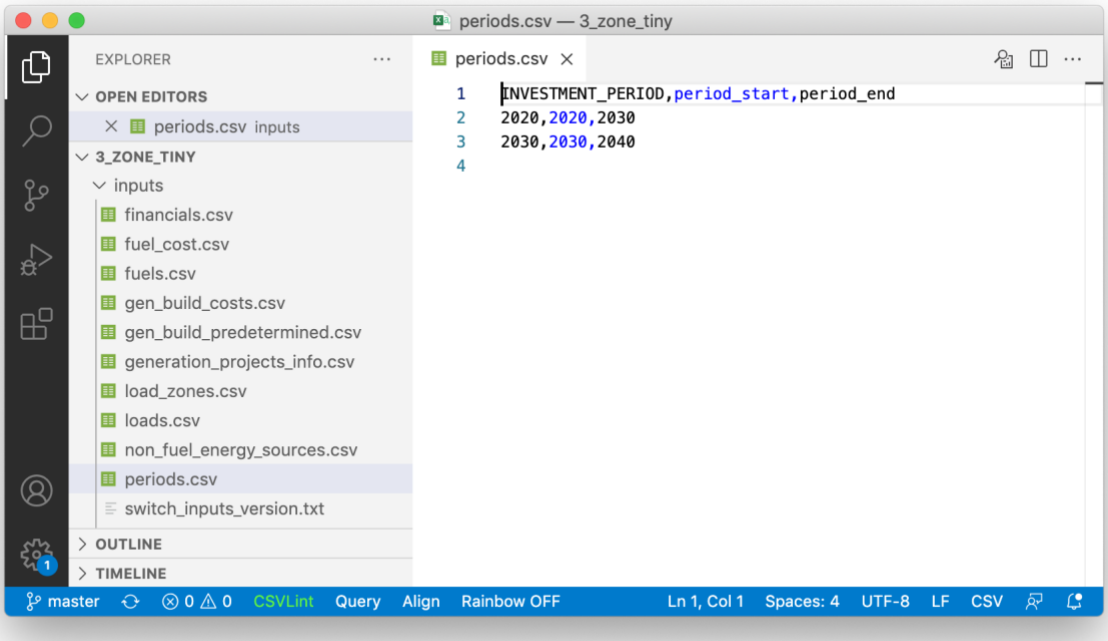
modules.txt lists all the Python modules that will be used to define your model. Some of these are required for every model: timescales, financials, load_zones, energy_sources.properties, generators.core.build, generators.core.dispatch, generators.core.no_commit (or an alternative) and energy_sources.fuel_costs.markets (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

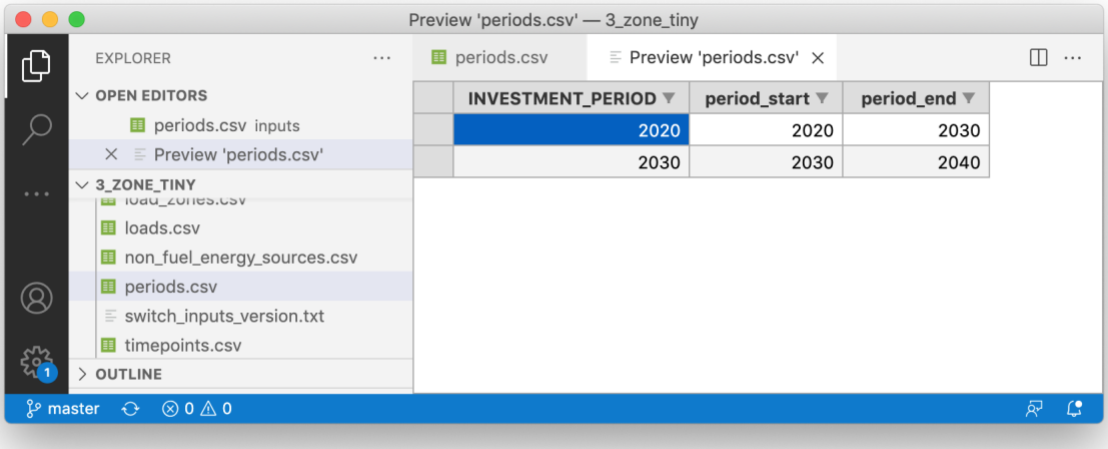
1.3.1 Opening .csv files in VS Code

For this tutorial, we will view and edit .csv files in VS Code, using the Rainbow CSV and Excel Viewer extensions 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 in your spreadsheet program.

> Click on the "inputs" folder to look at the files inside. Then double-click on "periods.csv". You should see a window similar to the one below.



> Click on the magnifying glass/box icon near the upper, right corner to open the file in a grid view instead. When you hover over this icon, you will see a "Open Preview" popup description. Alternatively, you can right-click on the periods.csv tab or file and choose "Open Preview". Then double-click on the vertical dividers between columns to see the whole titles. You will get a window like this:



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

The grid preview view provides simple filtering and sorting tools, which can be accessed by clicking on the gray triangles at the top of each column. The normal text view has some hidden features, e.g., an SQL-like query language that can be accessed via View > Command Palette… > Rainbow CSV: RBQL. You can find documentation for each extension by using View > Extensions, then clicking on the extension you are interested in.

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 values in the index columns 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 or first few columns 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. (period_end can be either 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.

gen_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 >= 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)
gen_is_baseload	identifies baseload power plants, which must run at a constant output level (0 or 1 value)
gen_is_cogen	(optional) not implemented in standard modules, but available for custom modules
gen_energy_source	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 gen_multiple_fuels.csv
gen_unit_size	(optional) fixed size for incremental capacity additions or unit commitment; has no effect unless generators.core.gen_build_discrete and/or generators.core.commit.discrete modules are loaded
gen_ccs_capture_efficiency	(optional) fraction of CO_2 emissions captured by CCS equipment
gen_ccs_energy_load	(optional) fraction of power production lost to CCS operation
gen_storage_efficiency	(optional) round-trip efficiency for storage projects
gen_store_to_release_ratio	(optional) storage power rating for a storage project, given as a ratio vs. the power output rating
gen_storage_energy_to_power_ratio	(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
gen_storage_max_cycles_per_year	(optional) limit on number of full charge/discharge cycles per year for storage projects
gen_min_uptime	(optional) minimum number of hours that a generator must remain committed (turned on) after it is first committed
gen_min_downtime	(optional) minimum number of hours that a generator must remain uncommitted after it is first decommitted (turned off)
gen_startup_fuel	(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 (build_gen_predetermined, in MW). Capacity specified here will automatically be retired after it reaches max_age_years.

gen_multiple_fuels.csv is only needed if some generators use multiple fuels. This file has two columns, generation_project and fuel. It should contain one row for every allowed combination of generation project and fuel, for all the multi-fuel projects. These projects should also show gen_energy_source = "multiple" in gen_info.csv.

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_2 intensity of each (tonnes CO_2 per million Btu combusted), upstream CO_2 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 gen_info.csv must be in either FUELS or NON_FUEL_ENERGY_SOURCES.

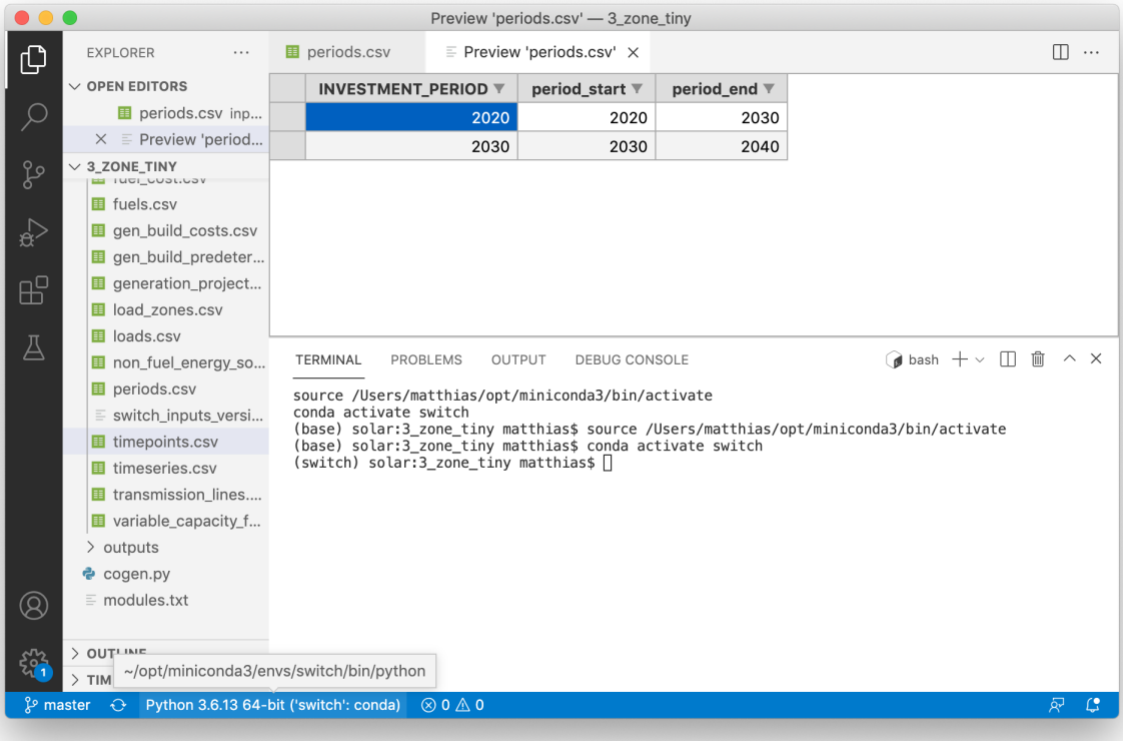
financials.csv defines financial parameters for the model. All costs are discounted to the base_financial_year using the discount_rate. Capital costs are amortized over the life of the project using the interest_rate (optional, defaults to discount_rate). Note that unlike the other parameters we have 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.

> In VS Code, choose View > Command Palette… > Python: Select Interpreter. Then choose the interpreter that mentions ('switch': conda). It should be in ~/opt/miniconda3/envs/switch/bin/python. You only need to do this the first time you open a folder in VS Code.

Next choose Terminal > New Terminal. This will open a terminal pane at the bottom of the VS Code window, where you can type commands to the operating system. You may want to make the window larger so you can see your files (top) and commands (bottom) at the same time. Your window should look something like this:



If you were unable to get conda to work in VS Code during the pre-tutorial setup, you should instead open Terminal.app (Mac) or Anaconda Command Prompt (Windows). Then run "conda activate switch", then use the "cd" command to navigate to the 3_zone_tiny directory. Then you can run all the boxed commands in this window.

Type this command in the terminal pane:

```
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`", "--stream-solver" and "`--sorted-output`" options (--verbose makes Switch show more messages, --stream-solver displays messages from the solver in addition to Switch, and --sorted-output ensures that data in the output files are sorted alphabetically).

Run this command:

```
switch solve --verbose --stream-solver --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 directory specified by `--outputs_dir`, which is "outputs" by default.

1.5 View outputs

Click on the "outputs" folder in the Explorer pane on the left. 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 the files 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 **total_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).

*** include other standard outputs
*** These are new:
outputs/dispatch_wide.csv
outputs/gen_build.csv
outputs/gen_project_annual_summary.csv
outputs/local_td_energy_balance.csv
outputs/local_td_energy_balance_wide.csv

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 VS Code, close any tabs that you are no longer using. Then right-click on the "inputs" folder in the Explorer pane and choose "Copy" (ctrl-click or two-fingered click on a Mac). Then scroll down to the gray area below the file and folders, right-click and choose "Paste". (Or you can use cmd-C (Mac) or ctrl-C (Windows) to copy the folder and cmd-P/ctrl-P to paste it.) Then right-click on the new directory and rename it to "inputs_new".

Open gen_build_costs.csv inside inputs_new. Find the C-Coal_IGCC project (rows 39 and 40). Type 6000000 in the gen_overnight_cost column (the third column) for both rows to change the capital cost for 2020 and 2030 vintages from $2,983,440/MW to $6,000,000/MW. (This should be done in the normal text view, not in the gridded preview window.)

Save the gen_build_costs.csv file.

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

Choose File > New File in VS Code. Then enter the following lines into the window:

```
--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.

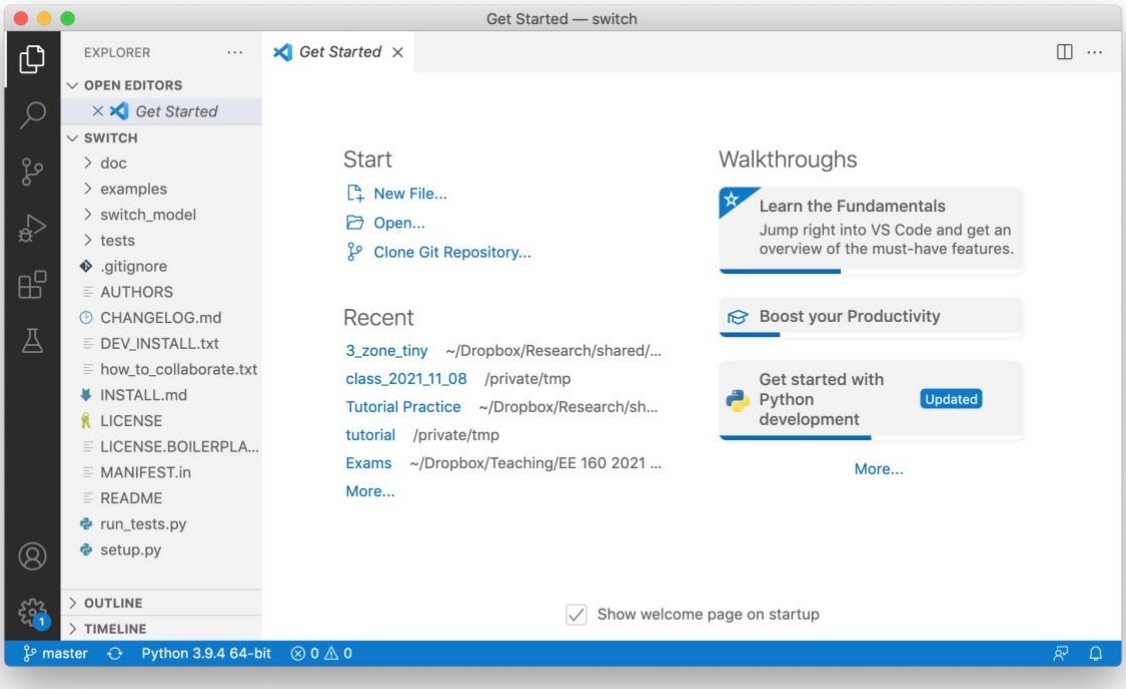
Now switch back to VS Code and open the versions of BuildGen.csv in the outputs and outputs_new folders. You should find that Switch has abandoned the plan to build C-Coal_IGCC in 2020 (0.0 instead of 11.12 MW before) and is now planning to build 11.4 MW of N-Coal_IGCC (formerly 0.0). You can use VS Code to compare the files by clicking on one in the Explorer pane, then holding down cmd (Mac) or ctrl (Windows) while clicking on the second one. If both files are open already, you can easily do this in the "Open Editors" section at the top of the Explorer pane. Then right click on one of the selected files and choose "Compare Selected".

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.

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 which file Switch will read this from. You may also want to check on how pre-existing transmission capacity is defined and incorporated into Switch. Here we will show how to do that.

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



> As an alternative, you can also browse the Switch source code by going to http://switch-model.org in a web browser.

If you browse through the "examples" directory, you will find a variety of examples that illustrate features of Switch. You can navigate to these in a terminal pane (with "cd") and use "switch solve" to solve each of them.[1] However, for this tutorial, we will focus only on the "switch_model" directory, where all the Switch source code resides.

> In the Explorer pane of VS Code, 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

[1] You will need to choose View > Command Palette... > Python: Select Interpreter > "Python <version> ('switch': conda)" in VS Code to set the interpreter for this folder at some point before you open the terminal pane.

module on its own. e.g., if you load switch_model.transmission.transport, Python will read the code in "switch_model/transmission/transport/__init__.py".

> Browse through build.py to see the documentation at the top and in the load_inputs() function.

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 function is not defined in the module, Switch skips it.

One of the first functions 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 this module 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 components onto this model. Later, Switch will load data into the model, solve the model and report the results. It does most of these stages by calling functions in the individual modules, as discussed below. 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 treated 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 the load_inputs() function in every module 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 Edit > Find in Files.) 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 the https://switch-model.org 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 Edit > Find 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. List those in modules.txt in your working directory.
- 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).

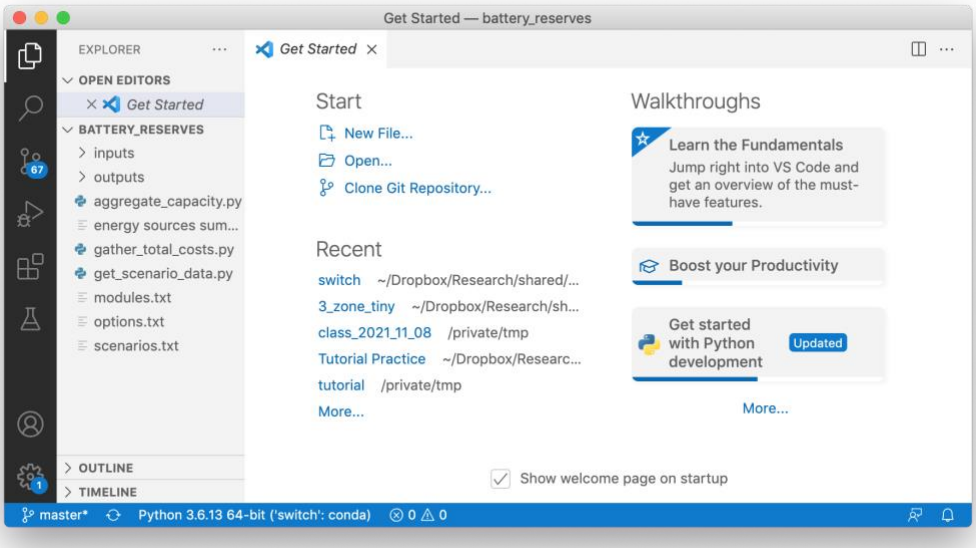
We are currently working on streamlining the source code and documentation, e.g., moving description, definition and file name together in define_components() and creating a website with descriptions of the columns in each table and hyperlinks to the definitions.

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 VS Code, use File > Open Folder… to open the battery_reserves folder inside the switch_tutorial folder that you created during the pre-tutorial setup. Then choose View > Command Palette… > Python: Select Interpreter > "Python <version> ('switch')" in VS Code to set the interpreter for this folder. You should get 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 separate 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 inside inputs/regulation (or one of the other inputs subdirectories). 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 several 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 the spinning_reserves_advanced module) 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 gen_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 open a terminal pane 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 VS Code. If the lines are wrapping, turn off View > Word 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. These are the details for each one:

- **battery_bulk:** 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:** 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:** 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:** 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**: 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 VS Code. 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. If you have installed COIN CBC, you could add "--solver cbc" to options.txt. 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 pane 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 terminal panes, navigate to the battery_reserves directory in each one, 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 can use the --log-run option to store the output for each scenario in a separate file. You can also use a different --logs-dir setting for each scenario to put the logs in specific directories.

These models are large and each take over an hour to solve with a commercial solver like gurobi or cplex. You are not likely to get answers in reasonable time with glpk or cbc. You can wait for these scenarios 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, we will use Microsoft Excel and minimal automation. You may prefer other tools or a mix of tools (we recommend Python with pandas and matplotlib, and R is another good option). If you have Excel installed on your own computer, you can follow along with that, or otherwise you can skip this section or use whatever tool you prefer for data visualization.

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 VS Code. 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	19562899003
battery bulk and conting	19449348811
battery bulk and reg	19185011329
dr bulk	18521344173
dr bulk and reserves	18480092053

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 "=C$2-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	19562899003	42528.0413	0
battery bulk and conting	19449348811	42281.1931	246.848242
battery bulk and reg	19185011329	41706.5464	821.494944
dr bulk	18521344173	40263.7917	2264.24963
dr bulk and reserves	18480092053	40174.1132	2353.92815

Inspecting this, we conclude that the savings are modest when bulk-storage batteries also provide contingency reserves ($247 over 30 years). They are somewhat higher when the batteries provide regulating reserves ($821 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—$2,264 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 $89 per customer, which may not be 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.)

There is also a script in the "battery_reserves" directory called "gather_total_costs.py" that automates the steps above. You can run this script by opening a terminal pane 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-O), then production from various technologies (columns P-X), then some information on curtailment, then total centralized and distributed production, then all types of loads (columns AE–AH), 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 900 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.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.

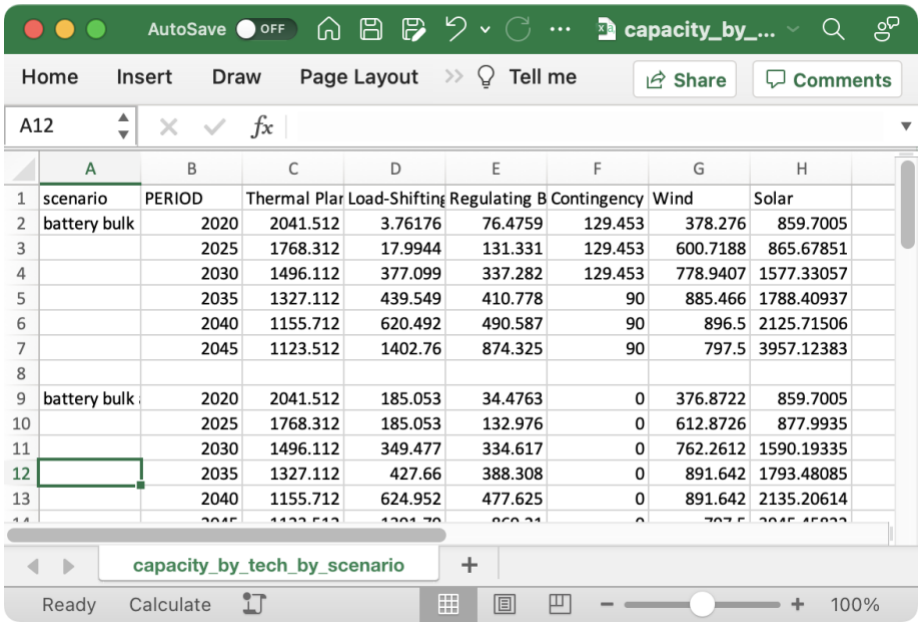
2.2.3 Compare capacity of different generation technologies across scenarios (advanced)

Switch back to the terminal pane you have open on the "reserve_study" directory. Then run the following script, which aggregates capacity investment data into a form that is easy to plot in Excel.

```
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 VS Code. (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 VS Code's Explorer pane, 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
1	scenario	PERIOD	Thermal Plar	Load-Shifting	Regulating B	Contingency	Wind	Solar
2	battery bulk	2020	2041.512	3.76176	76.4759	129.453	378.276	859.7005
3		2025	1768.312	17.9944	131.331	129.453	600.7188	865.67851
4		2030	1496.112	377.099	337.282	129.453	778.9407	1577.33057
5		2035	1327.112	439.549	410.778	90	885.466	1788.40937
6		2040	1155.712	620.492	490.587	90	896.5	2125.71506
7		2045	1123.512	1402.76	874.325	90	797.5	3957.12383
8								
9	battery bulk	2020	2041.512	185.053	34.4763	0	376.8722	859.7005
10		2025	1768.312	185.053	132.976	0	612.8726	877.9935
11		2030	1496.112	349.477	334.617	0	762.2612	1590.19335
12		2035	1327.112	427.66	388.308	0	891.642	1793.48085
13		2040	1155.712	624.952	477.625	0	891.642	2135.20614

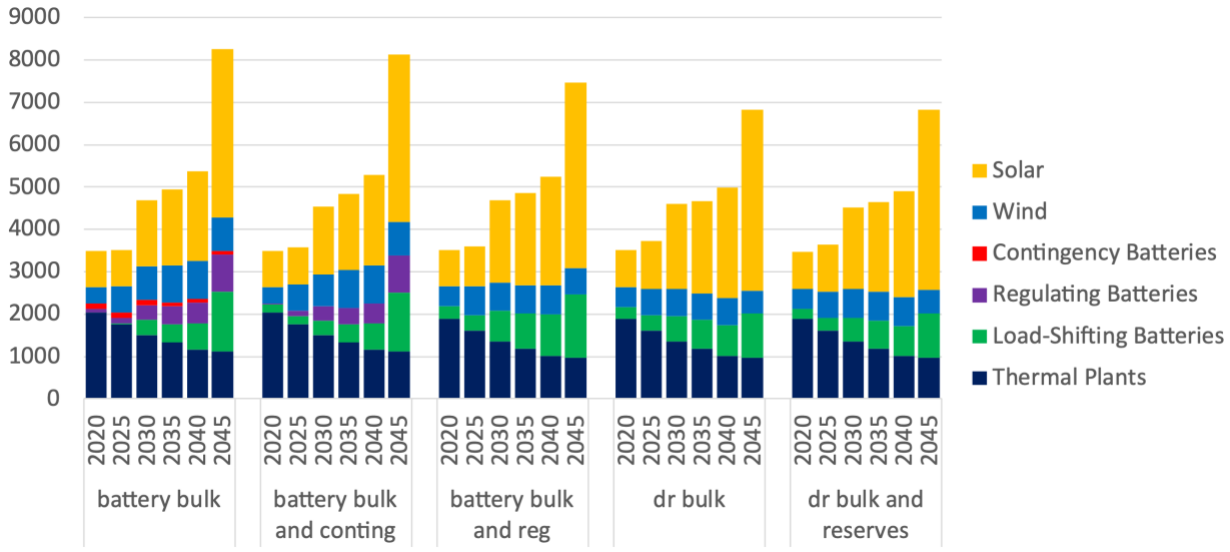
Next, select all the data cells and headers (C1:H35) but not the row labels. Then go to the Insert tab, click the dropdown arrow next to the column chart icon, and choose a 2D stacked column chart. You should then get a chart, and the Chart Design tab will activate.

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. We 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.
- Choose suitable colors for each series.

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 drop out when going from “battery bulk” to “battery bulk and conting”. Then regulating batteries drop out when going from “battery bulk and conting” to “battery bulk and reg” (both regulation and contingency reserves are now provided mainly by load-shifting batteries). 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 VS Code, use 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 in the Explorer pane 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, VS Code 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 VS Code, 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 VS Code (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).