Tools for Energy Model Optimization and Analysis (Temoa)

Release 2022-01-09

Joe DeCarolis, Kevin Hunter

Jan 09, 2022

CONTENTS

PREFACE

This manual, in both PDF and HTML form, is the official documentation of Tools for Energy Model Optimization and Analysis (Temoa). It describes all functionality of the Temoa model, and provides a mathematical description of the implemented equations.

Besides this documentation, there are a couple other sources for Temoa-oriented information. The most interactive is the mailing list, and we encourage any and all questions related to energy system modeling. Publications are good introductory resources, but are not guaranteed to be the most up-to-date as information and implementations evolve quickly. As with many software-oriented projects, even before this manual, *the code is the most definitive resource*. That said, please let us know (via the mailing list, or other avenue) of any discrepancies you find, and we will fix it as soon as possible.

1.1 What is Temoa?

Temoa is an energy system optimization model (ESOM). Briefly, ESOMs optimize the installation and utilization of energy technology capacity over a user-defined time horizon. Optimal decisions are driven by an objective function that minimizes the cost of energy supply. Conceptually, one may think of an ESOM as a "left-to-right" network graph, with a set of energy sources on the lefthand side of the graph that are transformed into consumable energy commodities by a set of energy technologies, which are ultimately used to meet demands on the righthand side of the network graph.[4]

Key features of the core Temoa model include:

- Flexible time slicing by season and time-of-day

- Variable length model time periods

- Technology vintaging

- Separate technology loan periods and lifetimes

- Global and technology-specific discount rates

- Capability to perform stochastic optimization

- Capability to perform modeling-to-generate alternatives (MGA)

Temoa design features include:

- Source code licensed under GPLv2, available through Github[1]

- Open source software stack

---

[4] For a more in-depth description of energy system optimization models (ESOMs) and guidance on how to use them, please see: DeCarolis et al. (2017) "Formalizing best practice for energy system optimization modelling", Applied Energy, 194: 184-198.

[1] The two main goals behind Temoa are transparency and repeatability, hence the GPLv2 license. Unfortunately, there are some harsh realities in the current climate of energy modeling, so this license is not a guarantee of openness. This documentation touches on the issues involved in the final section.

- Part of a rich Python ecosystem

- Data stored in a relational database system (sqlite)

- Ability to utilize multi-core and compute cluster environments

The word 'Temoa' is actually an acronym for "Tools for Energy Model Optimization and Analysis," currently composed of four (major) pieces of infrastructure:

- The mathematical model

- The implemented model (code)

- Surrounding tools

- An online presence

Each of these pieces is fundamental to creating a transparent and usable model with a community oriented around collaboration.

1.2 Why Temoa?

In short, because we believe that ESOM-based analyses should be repeatable by independent third parties. The only way to make this happen is to have a freely available model, and to create an ecosystem of freely shared data and model inputs.

For a longer explanation, please see [DeCarolisHunterSreepathi13] (available from the project website. In summary, ESOM-based analyses are (1) impossible to validate, (2) complex enough as to be non-repeatable without electronic access to **exact** versions of code *and* data input, and (3) often do a poor job addressing uncertainty. We believe that ESOM-based analyses should be completely open, independently reproducible, electronically available, and address uncertainty about the future.

1.3 Temoa Origin and Pronunciation

While we use 'Temoa' as an acronym, it is an actual word in the Nahuatl (Aztec) language, meaning "to seek something."

One pronounces the word 'Temoa' as "teh", "moe", "uh". Though TEMOA is an acronym for 'Tools for Energy Model Optimization and Analysis', we generally use 'Temoa' as a proper noun, and so forgo the need for all-caps.

1.4 Bug Reporting

Temoa strives for correctness. Unfortunately, as an energy system model and software project there are plenty of levels and avenues for error. If you spot a bug, inconsistency, or general "that could be improved", we want to hear about it.

If you are a software developer-type, feel free to open an issue on our GitHub Issue tracker. If you would rather not create a GitHub account, feel free to let us know the issue on our mailing list.

QUICK START

2.1 Installing Software Elements

Temoa is implemented in Pyomo, which is in turn written in Python. Consequently, Temoa will run on Linux, Mac, Windows, or any operating system that Pyomo supports. There are several open source software elements required to run Temoa. The easiest way to install these elements is to create a conda environment in which to run the model. Creating a customized environment ensures that the latest version of Temoa is compatible with the required software elements. To begin, you need to have conda installed either via miniconda or anaconda. Next, download the environment.yml file from our Github repo, and place it in a new directory named 'temoa-py3.' Create this new directory in a location where you wish to store the environment. On a MAc, the anaconda environment files are stored in the following location by default: `<user>/opt/anaconda/envs` Navigate to this directory and execute the following from the command line:

```
$ conda env create
```

Then activate the environment as follows:

```
$ conda activate temoa-py3
```

For additional guidance, *This YouTube tutorial <https://youtu.be/XYoxUGuZG2A>* walks through the creation of the Temoa environment. More information on virtual environments can be found here. This new conda environment contains several elements, including Python 3, a compatible version of Pyomo, matplotlib, numpy, scipy, and two free solvers (GLPK and CBC). Windows users: the CBC solver is not available for Windows through conda. Thus, in order to install the environment properly, the last line of the `environment.yml` file specifying `coincbc` should be deleted. A few notes for on the choice of solvers. Different solvers have widely varying solution times. If you plan to run Temoa with large datasets and/or conduct uncertainty analysis, you may want to consider installing commercial linear solvers such as CPLEX or Gurobi. Both offer free academic licenses.

There are three ways to run the model, each of which is detailed below. Note that the example commands utilize 'temoa_utopia', a commonly used test case for ESOMs.

2.2 Obtaining Temoa

Now that you have functioning environment, you need to obtain the source code for Temoa. There are a couple of options for obtaining and running Temoa from GitHub. If you want to simply run the model, you can download Temoa from GitHub as a zip file. Navigate to our Github repo, and click the green 'clone or download' button near the top-right corner. Select 'Download ZIP,' and you can download the entire Temoa `energysystem` (our main branch) to your local machine. The second option creates a local copy of the model source code in our GitHub repository. This is a two step process: first install git and then 'clone' the repository. Under Linux, git can be installed through the default

package manager. Git for Windows and Mac can be downloaded from the Git website. To clone the Temoa repository, navigate to the directory where you want the model to reside and type the following from the prompt:

```
$ git clone https://github.com/TemoaProject/temoa/
```

Note that cloning the repository will supply the latest version of the code, and allow you to archive changes to the code and data in your own local git repository.

A few basic input data files are included in the `temoa/data_files` folder. Additional Temoa-compatible datasets are available in this separate GitHub repo.

The installation procedures above are meant to be generic and should work across different platforms. Nonetheless, system-specific ambiguities and unexpected conditions inevitably arise. Please use the Temoa forum to ask for help.

2.3 Running Temoa

Temoa should always be run from the top-level from the top-level `temoa` directory. The most basic way to run Temoa is with an input data (DAT) file:

```
$ python temoa_model/ /path/to/dat/file
```

This option will simply run the model and output the results to the shell. To make sure the model is functioning correctly, try running with the 'Utopia' dataset:

```
$ python temoa_model/ data_files/utopia-15.dat
```

To run the model with more features, use a configuration ('config') file. An example config file called `config_sample` resides within the `temoa_model` folder. Running the model with a config file allows the user to (1) use a sqlite database for storing input and output data, (2) create a formatted Excel output file, (2) specify the solver to use, (3) return the log file produced during model execution, (4) return the lp file utilized by the solver, and (5) to execute modeling-to-generate alternatives (MGA).

```
$ python temoa_model/ --config=temoa_model/config_sample
```

For general help, use –help:

```
$ python  temoa_model/  --help
usage: temoa_model [-h] [--path_to_logs PATH_TO_LOGS] [--config CONFIG]
                   [--solver {bilevel_blp_global,bilevel_blp_local,bilevel_ld,cplex,mpec_
↪minlp,mpec_nlp,openopt,ps} ]
                   [dot_dat [dot_dat ...]]

positional arguments:
  dot_dat               AMPL-format data file(s) with which to create a model
                        instance. e.g. "data.dat"

optional arguments:
  -h, --help            show this help message and exit
  --path_to_logs PATH_TO_LOGS
                        Path to where debug logs will be generated by default.
                        See folder debug_logs in data_files.
  --config CONFIG       Path to file containing configuration information.
  --solver {bilevel_blp_global,bilevel_blp_local,bilevel_ld,cplex,mpec_minlp,mpec_nlp,
↪openopt,ps}
                        Which backend solver to use. See 'pyomo --help-
```

```
                         solvers' for a list of solvers with which Pyomo can
                         interface. The list shown here is what Pyomo can
                         currently find on this system. [Default: cplex]
```

To supplement this documentation, we have also created a *YouTube video tutorial <https://youtu.be/WtzCrroAXnQ>* that explains how to run Temoa from the command line. There is also an option to run *Temoa on the cloud <https://model.temoacloud.com>*, which is explained in *this video tutorial <https://youtu.be/fxYO_kIs364>*.

DATABASE CONSTRUCTION

Input datasets in Temoa can be constructed either as text files or relational databases. Input text files are referred to as 'DAT' files and follow a specific format. Take a look at the example DAT files in the `temoa/data_files` directory.

While DAT files work fine for small datasets, relational databases are preferred for larger datasets. To first order, you can think of a database as a collection of tables, where a 'primary key' within each table defines a unique entry (i.e., row) within the table. In addition, a 'foreign key' defines a table element drawn from another table. Foreign keys enforce the defined relationships between different sets and parameters.

Temoa uses sqlite, a widely used, self-contained database system. Building a database first requires constructing a sql file, which is simply a text file that defines the structure of different database tables and includes the input data. The snippet below is from the technology table used to define the 'temoa_utopia' dataset:

```
CREATE TABLE technologies (
tech text primary key,
flag text,
sector text,
tech_desc text,
tech_category text,
FOREIGN KEY(flag) REFERENCES technology_labels(tech_labels),
FOREIGN KEY(sector) REFERENCES sector_labels(sector));
INSERT INTO "technologies" VALUES('IMPDSL1','r','supply',' imported diesel','petroleum');
INSERT INTO "technologies" VALUES('IMPGSL1','r','supply',' imported gasoline','petroleum
↪');
INSERT INTO "technologies" VALUES('IMPHCO1','r','supply',' imported coal','coal');
```

The first line creates the table. **Lines 2-6** define the columns within this table. Note that the the technology ('tech') name defines the primary key. Therefore, the same technology name cannot be entered twice; each technology name must be unique. **Lines 7-8** define foreign keys within the table. For example, each technology should be specified with a label (e.g., 'r' for 'resource'). Those labels must come from the 'technology_labels' table. Likewise, the sector name must be defined in the 'sector_labels' table. This enforcement of names across tables using foreign keys helps immediately catch typos. (As you can imagine, typos happen in plain text files and Excel when defining thousands of rows of data.) Another big advantage of using databases is that the model run outputs are stored in separate database output tables. The outputs by model run are indexed by a scenario name, which makes it possible to perform thousands of runs, programatically store all the results, and execute arbitrary queries that instantaneously return the requested data.

Because some database table elements serve as foreign keys in other tables, we recommend that you populate input tables in the following order:

Group 1: labels used for internal database processing

- commodity labels: Need to identify which type of commodity. Feel free to change the abbreviations.

- technology labels: Need to identify which type of technology. Feel free to change the abbreviations.

- time_period_labels: Used to distinguish which time periods are simply used to specify pre-existing vintages and which represent future optimization periods.

Group 2: sets used within Temoa

- commodities: list of commodities used within the database

- technologies: list of technologies used within the database

- time_periods: list of both past and future time periods considered in the database

- time_season: seasons modeled in the database

- time_of_day: time of day segments modeled in the database

Group 3: parameters used to define processes within Temoa

- GlobalDiscountRate

- Demand

- DemandSpecificDistribution

- Efficiency

- ExistingCapacity

- CapacityFactor

- CapacityFactorProcess (only if CF varies by vintage; overwrites CapacityFactor)

- Capacity2Activity

- CostFixed

- CostInvest

- CostVariable

- EmissionsActivity

- LifetimeLoanTech

- LifetimeProcess

- LifetimeTech

Group 4: parameters used to define constraints within Temoa

- GrowthRateSeed

- GrowthRateMax

- MinCapacity

- MaxCapacity

- MinActivity

- MaxActivity

- RampUp

- RampDown

- TechOutputSplit

- TechInputSplit

For help getting started, take a look at how `data_files/temoa_utopia.sql` is constructed. Use `data_files/temoa_schema.sql` (a database file with the requisite structure but no data added) to begin building your own database file. We recommend leaving the database structure intact, and simply adding data to the schema file. Once the sql file is complete, you can convert it into a binary sqlite file by installing sqlite3 and executing the following command:

```
$ sqlite3 my_database.sqlite < my_database.sql
```

Now you can specify this database as the source for both input and output data in the config file.

VISUALIZATION

4.1 Network Diagrams

Since Temoa model consists of an energy network in which technologies are connected by the flow of energy commodities, a directed network graph represents an excellent way to visualize a given energy system representation in a Temoa-compatible input database. Temoa utilizes an open source graphics package called Graphviz to create a series of data-specific and interactive energy-system maps. Currently, the output graphs consist of a full energy system map as well as capacity and activity results per model time period. In addition, users can create subgraphs focused on a particular commodity or technology.

There are a couple ways to utilize Graphviz. The first way is to use the version embedded in the `Network_diagrams.ipynb` jupyter notebook available in the `data_processing` folder. This *YouTube video tutorial <https://youtu.be/06SMhFilwLo>* walks through the process of using the jupyter notebook.

The second way is to use graphviz from the command line. To do so, navigate to the `data_processing` folder, where the graphviz script resides. To review all of the graphviz options, use the `--help` flag:

```
$ python MakeGraphviz.py --help
```

The most basic way to use graphviz is to view the full energy system map:

```
$ python MakeGraphviz.py -i ../data_files/temoa_utopia.sqlite
```

In the command above, note that we have to point the Graphviz module to the `temoa_utopia` database file, which resides in the `data_files` directory. The resultant system map will look like this:

It is also possible to create a system map showing the optimal installed capacity and technology flows in a particular model time period. These results are associated with a specific model run stored in the model database. To view the results, include the scenario flag (`-s`) and a specific model year (`-y`).

Note that when Graphiz runs, it creates a folder within the `data_processing` folder. The folder itself is assigned the name of the database file, with `input_graphviz` appended to the end. This descriptor changes if using Graphviz to visualize output graphics. Within this Graphviz-generated folder are two files. The graphics file (default: svg) is a viewable image of the network. The dot file is the input file to Graphviz that is created programmatically. Note that the dot files provide another means to debug the model and create an archive of visualizations for auditing purposes. In addition, we have taken care to make these intermediate files well-formatted.

```
$ python MakeGraphviz.py -i ../data_files/temoa_utopia.sqlite -s test_run -y 1990
```

The output can also be fine-tuned to show results associated with a specific commodity or technology. For example:

```
$ python MakeGraphviz.py -i ../data_files/temoa_utopia.sqlite -s test_run -y 2010 -b E31
```
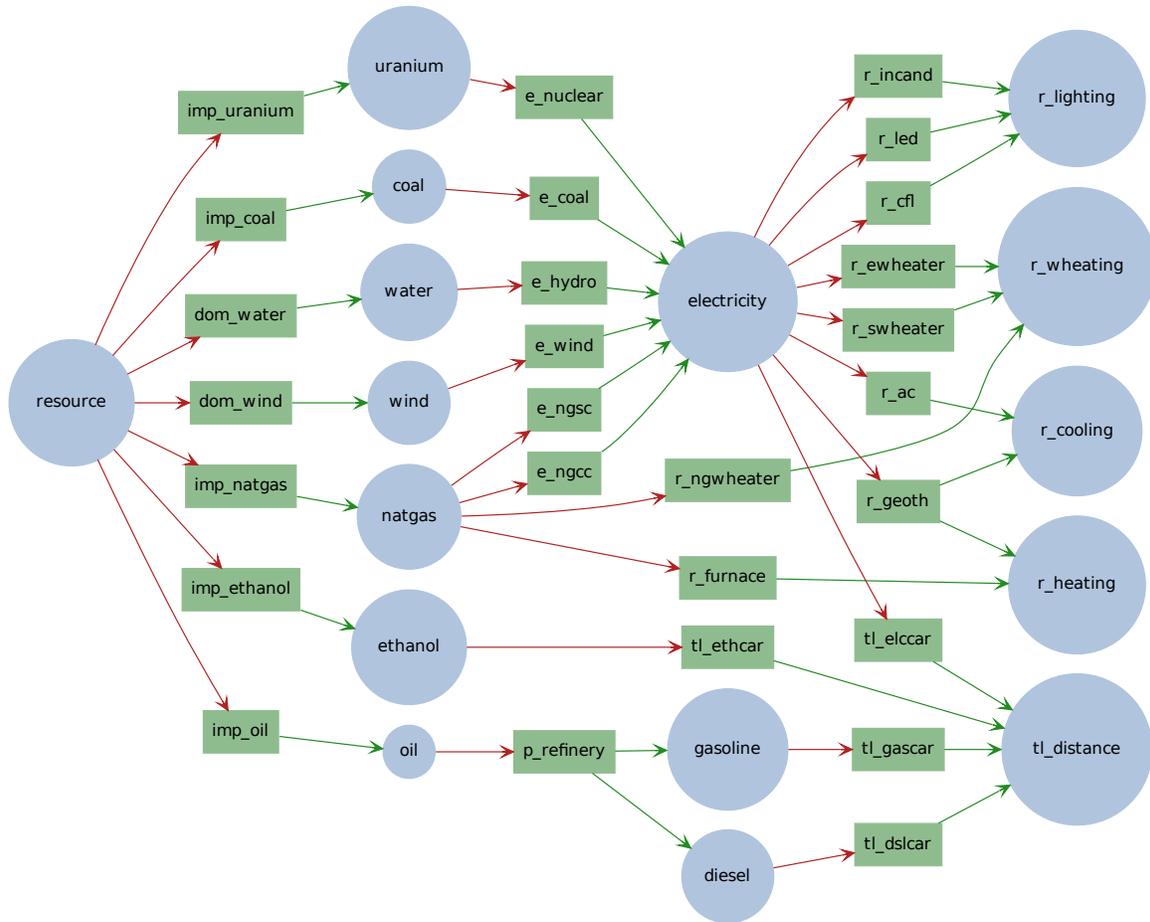
Fig. 1: This is a map of the simple 'Utopia' system, which we often use for testing purposes. The map shows the possible commodity flows through the system, providing a comprehensive overview of the system. Creating the simple system map is useful for debugging purposes in order to make sure that technologies are linked together properly via commodity flows.

Fig. 2: This graph shows the optimal installed capacity and commodity flows from the 'utopia' test system in 2010.

Fig. 3: In this case, the graph shows the commodity flow in and out of technology 'E31' in 2010, which is from the 'test_run' scenario drawn from the 'temoa_utopia' database.

4.2 Output Graphs

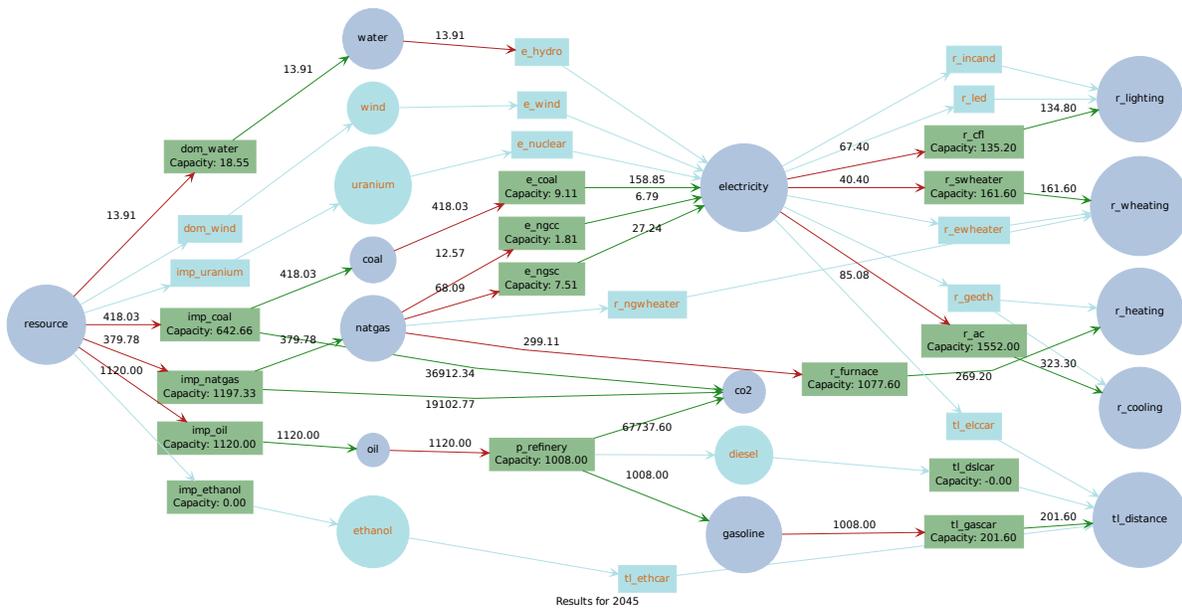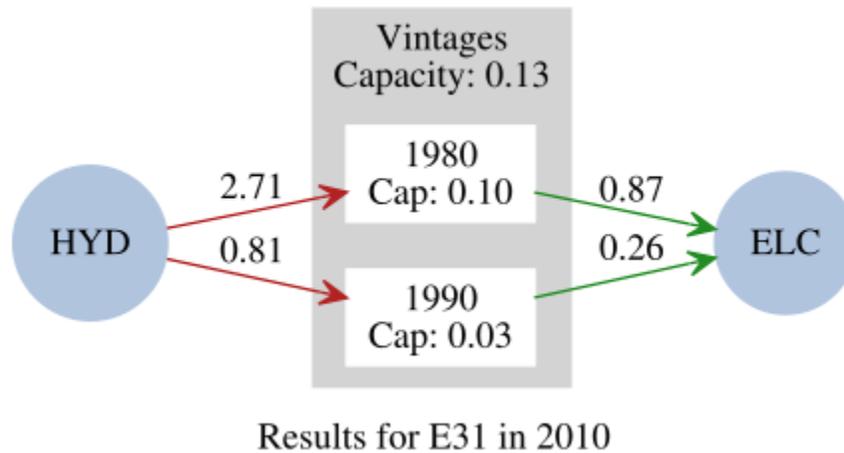Temoa can also be used to generate output graphs using *matplotlib <https://matplotlib.org/>*. From the command line, navigate to the `data_processing` folder and execute the following command:

```
$ python MakeOutputPlots.py --help
```

The command above will specify all of the flags required to created a stacked bar or line plot. For example, consider the following command:

```
$ python MakeOutputPlots.py -i ../data_files/temoa_utopia.sqlite -s test_run -p capacity␣
↪-c electric --super
```

Here is the result:
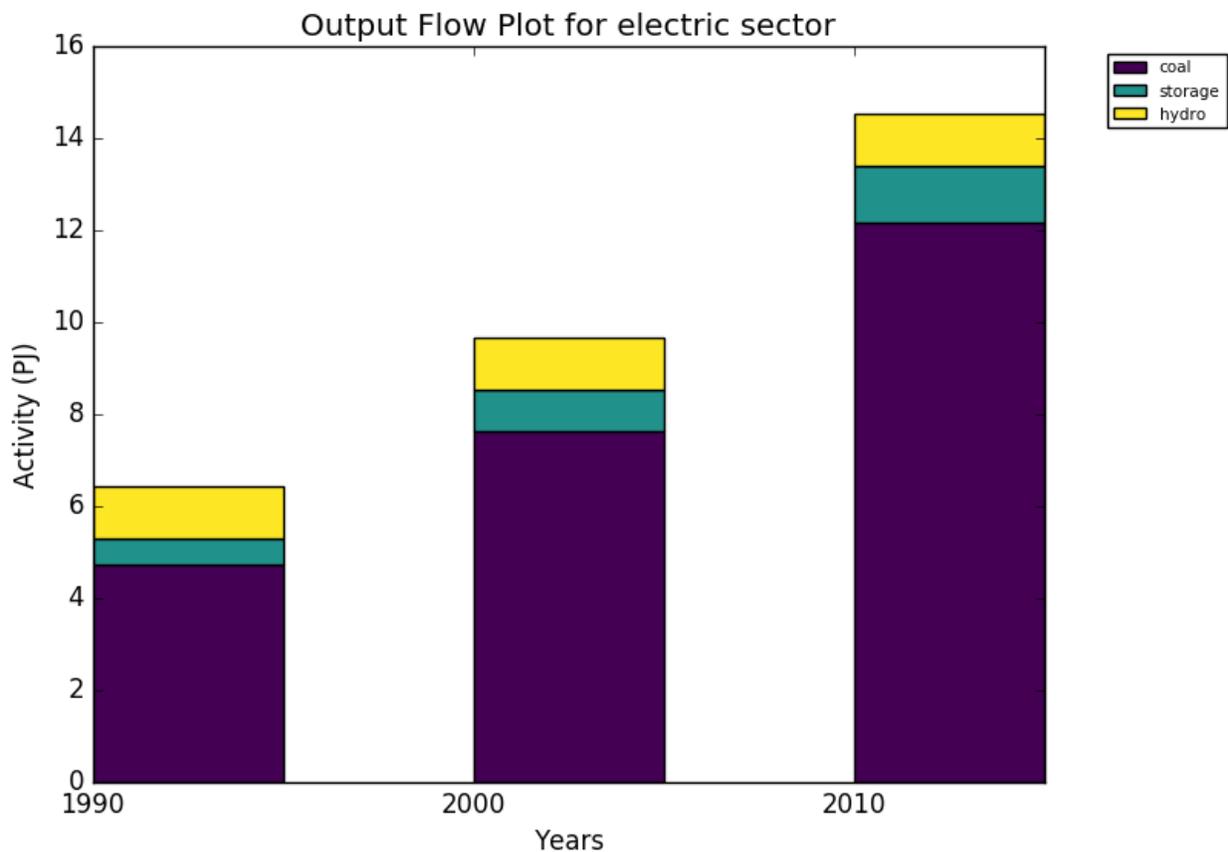

Fig. 4: This stacked bar plot represents the activity (i.e., output commodity flow) associated with each technology in the electric sector from the 'test_run' scenario drawn from the 'temoa_utopia' database. Because the `super` flag was specified, technologies are grouped together based on user-specified categories in the `tech_category`` column of the `technologies` table of the database.

THE MATH BEHIND TEMOA

To understand this section, the reader will need at least a cursory understanding of mathematical optimization. We omit here that introduction, and instead refer the reader to various available online sources. Temoa is formulated as an algebraic model that requires information organized into sets, parameters, variables, and equation definitions.

The heart of Temoa is a technology explicit energy system optimization model. It is an algebraic network of linked processes – where each process is defined by a set of engineering characteristics (e.g. capital cost, efficiency, capacity factor, emission rates) – that transform raw energy sources into end-use demands. The model objective function minimizes the present-value cost of energy supply by optimizing installed capacity and its utilization over time.
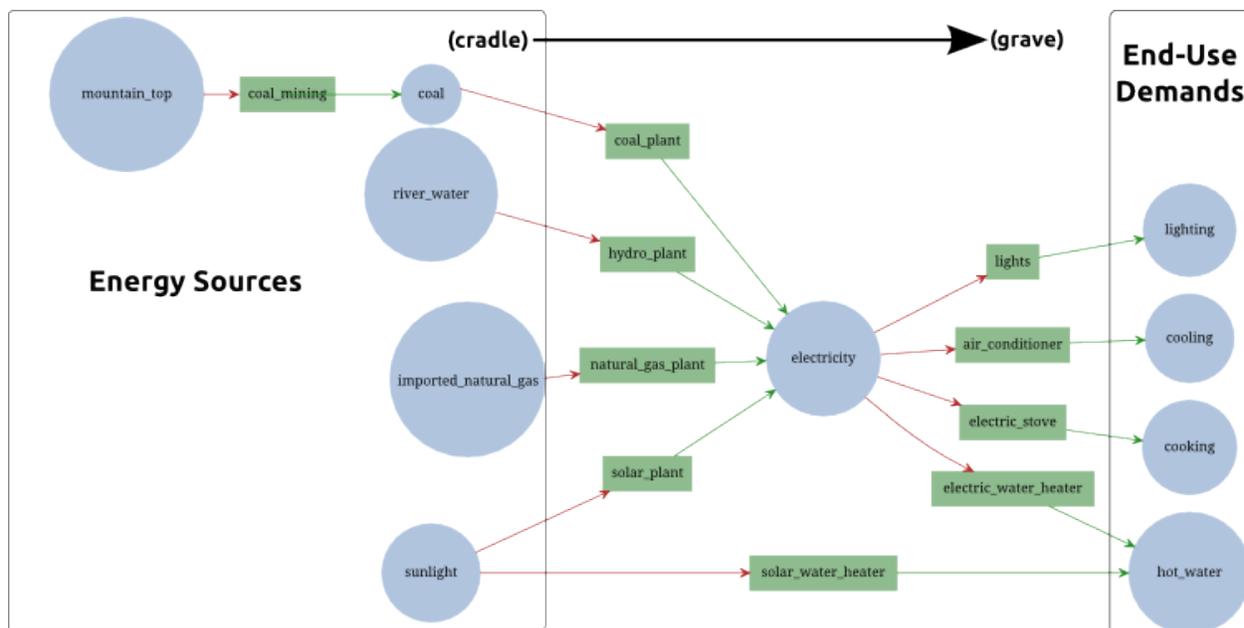
Fig. 1: A common visualization of energy system models is a directed network graph, with energy sources on the left and end-use demands on the right. The modeler must specify the end-use demands to be met, the technologies defined within the system (rectangles), and the inputs and outputs of each (red and green arrows). The circles represent distinct energy carriers that connect technologies within the energy system network.

The most fundamental tenet of the model is the understanding of energy flow, treating all processes as black boxes that take inputs and produce outputs. Specifically, Temoa does not care about the inner workings of a process, only its global input and output characteristics. In this vein, the above graphic can be broken down into process-specific elements. For example, the coal power plant takes as input coal and produces electricity, and is subject to various costs (e.g. variable costs) and constraints (e.g. efficiency) along the way.
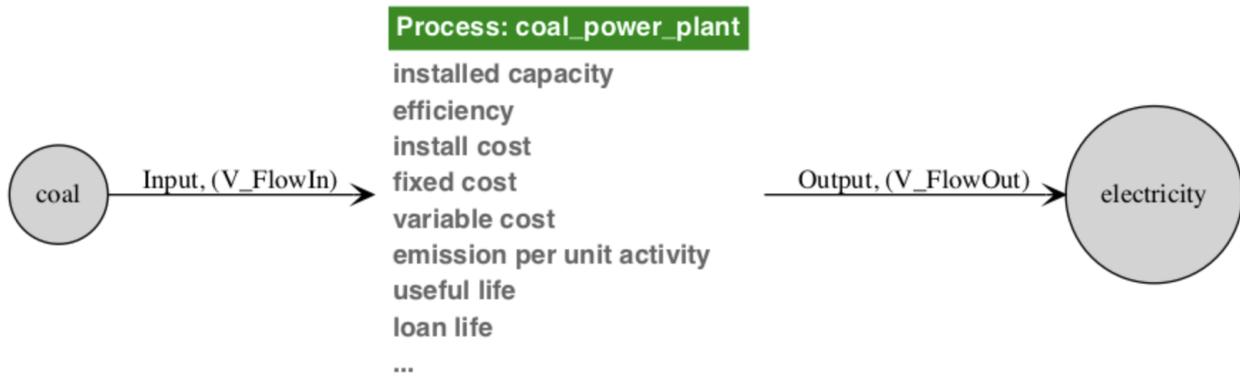
The modeler defines the processes and engineering characteristics through a combination of sets and parameters, described in the next few sections. Temoa then utilizes these parameters, along with the associated technology-specific decision variables for capacity and activity, to create the objective function and constraints that are used during the optimization process.

5.1 Conventions

- In the mathematical notation, we use CAPITALIZATION to denote a container, like a set, indexed variable, or indexed parameter. Sets use only a single letter, so we use the lower case to represent an item from the set. For example, T represents the set of all technologies and t represents a single item from T.

- Variables are named V_VarName within the code to aid readability. However, in the documentation where there is benefit of italics and other font manipulations, we elide the 'V_' prefix.

- In all equations, we **bold** variables to distinguish them from parameters. Take, for example, this excerpt from the Temoa default objective function:

$$C_{variable} = \sum_{p,s,d,i,t,v,o \in \Theta_{VC}} \left(VC_{p,t,v} \cdot R_p \cdot \mathbf{FO}_{p,s,d,i,t,v,o} \right)$$

Note that $C_{variable}$ is not bold, as it is a temporary variable used for clarity while constructing the objective function. It is not a structural variable and the solver never sees it.

- Where appropriate, we put the variable on the right side of the coefficient. In other words, this is not a preferred form of the previous equation:

$$C_{variable} = \sum_{r,p,s,d,i,t,v,o \in \Theta_{VC}} \left(\mathbf{FO}_{r,p,s,d,i,t,v,o} \cdot VC_{r,p,t,v} \cdot R_p \right)$$

- We generally put the limiting or defining aspect of an equation on the right hand side of the relational operator, and the aspect being limited or defined on the left hand side. For example, equation (5.1) defines Temoa's mathematical understanding of a process capacity (**CAP**) in terms of that process' activity (**ACT**):

$$\left(\mathrm{CFP}_{r,t,v} \cdot \mathrm{C2A}_{r,t} \cdot \mathrm{SEG}_{s,d} \cdot \mathrm{TLF}_{r,p,t,v} \right) \cdot \mathbf{CAP}_{r,t,v} = \sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{I,O} \mathbf{CUR}_{r,p,s,d,i,t,v,o}$$

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\mathrm{FO}}$$

- We use the word 'slice' to refer to the tuple of season and time of day $\{s, d\}$. Note that these time slices are user-defined, and can represent time ranging large blocks of time (e.g., winter-night) to every hour in a given season.

- We use the word 'process' to refer to the tuple of technology and vintage ($\{t, v\}$). For example, solar PV (technology) installed in 2030 (vintage).

- Mathematical notation:

    - We use the symbol $\mathbb{I}$ to represent the unit interval ([0, 1]).

    - We use the symbol $\mathbb{Z}$ to represent "the set of all integers."

    - We use the symbol $\mathbb{N}$ to represent natural numbers (i.e., integers greater than zero: 1, 2, 3, . . .).

    - We use the symbol $\mathbb{R}$ to denote the set of real numbers, and $\mathbb{R}_0^+$ to denote non-negative real numbers.

5.2 Sets

Table 1: List of all Temoa sets with which a modeler might interact. The asterisked (*) elements are automatically derived by the model and are not user-specifiable.

Set	Temoa Name	Data Type	Short Description
*C	`commodity_all`	string	union of all commodity sets
C^d	`commodity_demand`	string	end-use demand commodities
C^e	`commodity_emissions`	string	emission commodities (e.g. CO_2 NO_x)
C^p	`commodity_physical`	string	general energy forms (e.g. electricity, coal, uranium, oil)
*C^c	`commodity_carrier`	string	physical energy carriers and end-use demands ($C_p \cup C_d$)
I		string	alias of C^p; used in documentation only to mean "input"
O		string	alias of C^c; used in documentation only to mean "output"
P^e	`time_existing`	$\mathbb{Z}$	model periods before optimization begins
P^f	`time_future`	$\mathbb{Z}$	model time scale of interest; the last year is not optimized
*P^o	`time_optimize`	$\mathbb{Z}$	model time periods to optimize; ($P^f - \max(P^f)$)
R	`regions`	string	distinct geographical regions
*V	`vintage_all`	$\mathbb{Z}$	possible tech vintages; ($P^e \cup P^o$)
S	`time_season`	string	seasonal divisions (e.g. winter, summer)
D	`time_of_day`	string	time-of-day divisions (e.g. morning)
*T	`tech_all`	string	all technologies to be modeled; ($T^r \cup T^p$)
T^a	`tech_annual`	string	technologies that produce constant annual output; ($T^a \subset T$)
T^b	`tech_baseload`	string	baseload electric generators; ($T^b \subset T$)
T^c	`tech_curtailment`	string	technologies with curtailable output and no upstream cost; ($T^c \subset T$)
T^e	`tech_exchange`	string	technologies used for interregional commodity flow; ($T^e \subset T$)
T^f	`tech_flex`	string	technologies producing excess commodity flows; ($T^f \subset T$)
	`groups`	string	defined groups of technologies
T^f	`tech_groups`	string	technologies belonging to each group defined above
T^p	`tech_production`	string	techs producing intermediate commodities

Table 1 – continued from previous page

Set	Temoa Name	Data Type	Short Description
T^r	`tech_resource`	string	resource extraction technologies
T^m	`tech_ramping`	string	electric generators with a ramp rate limit; $(T^m \subset T)$
T^{res}	`tech_reserve`	string	electric generators contributing to the reserve margin requirement; $(T^e \subset T)$
T^s	`tech_storage`	string	storage technologies; $(T^s \subset T)$
T^v	`tech_variable`	string	technologies used in TechInputSplitAverage constraint; $(T^v \subset T)$
T^{cmax}	`tech_capacity_max`	string	subset of technologies used in MaxCapacitySet constraint; $(T^{cmax} \subset T)$
T^{cmin}	`tech_capacity_min`	string	subset of technologies used in MinCapacitySet constraint; $(T^{cmin} \subset T)$

Temoa uses two different set notation styles, one for code representation and one that utilizes standard algebraic notation. For brevity, the mathematical representation uses capital letters to denote sets, and lower case letters to represent items within sets. For example, T represents the set of all technologies and t represents an item within T.

The code representation is more verbose than the algebraic version, using full words. This documentation presents them in an italicized font. The same example of all technologies is represented in the code as `tech_all`. *Table 1* lists all of the Temoa sets, with both sets of notation.

There are four basic set "groups" within Temoa: periods, sub-annual "time slices", technologies, and energy commodities. The technology-related sets contain all the possible energy technologies that the model may build and the commodities sets contain all the input and output forms of energy that technologies consume and produce. The period and time slice sets merit a slightly longer discussion.

Temoa's conceptual model of *time* is broken up into three levels, and energy supply and demand is balanced at each of these levels:

- **Periods** - consecutive blocks of years, marked by the first year in the period. For example, a two-period model might consist of $P^f = \{2010, 2015, 2025\}$, representing the two periods of years from 2010 through 2014, and from 2015 through 2024. Note the that last period element (2025) does not represent a new time period, but rather defines the upper bound for the second time period.

- **Seasonal** - Each year may have multiple seasons. In a conventional time-sliced model, the seasons may typically represent the four seasons: winter, spring summer, and fall. However, the seasonal slices can represent any amount of time at the sub-period scale. For example, in a database with representative days, the seasonal slices can be used as generic containers to represent blocks of days.

- **Daily** - Within a season, a given day can be further subdivided into different time segments. Less detailed databases may include larger blocks of time, such as morning, afternoon, and night. In a database with representative days, each daily segment can be used to represent every hour of the day.

There are two specifiable period sets: `time_exist` (P^e) and `time_future` (P^f). The `time_exist` set contains periods before `time_future`. Its primary purpose is to specify the vintages for capacity that exist prior to the model optimization. The `time_future` set contains the future periods that the model will optimize. As this set must contain only integers, Temoa interprets the elements to be the boundaries of each period of interest. Thus, this is an ordered set and Temoa uses its elements to automatically calculate the length of each optimization period; modelers may exploit this to create variable period lengths within a given input database. Temoa "names" each optimization period by the first year, and makes them easily accessible via the `time_optimize` set. This final "period" set is not user-specifiable, but is an exact duplicate of `time_future`, less the largest element. In the above example, since $P^f = \{2010, 2015, 2025\}$, `time_optimize` does not contain 2025: $P^o = \{2010, 2015\}$.

One final note on periods: rather than optimizing each year within a period individually, Temoa makes the simplifying assumption that each time period contains n copies of a single, representative year. Temoa optimizes capacity and activity for just this characteristic year within each time period, assuming the results for different years in the same time

period are identical. The Temoa objective function, however, accounts for the total cost across all years in all model time periods. Figure 3.3 gives a graphical explanation of the annual delineation.
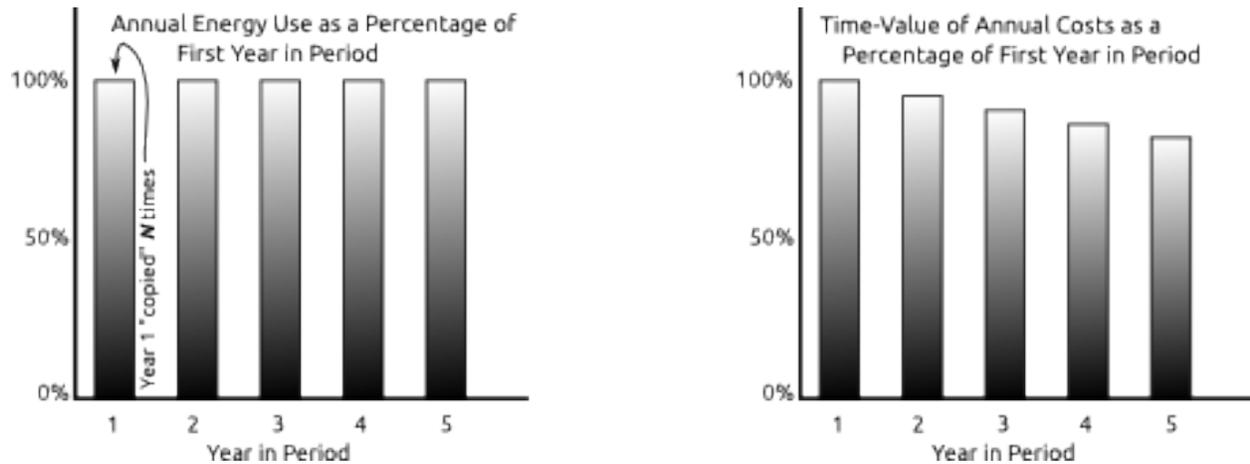

Fig. 2: The left graph is of energy, while the right graph is of the annual costs. The energy used in a period by a process is the same for all years (with exception for those processes that cease their useful life mid-period). However, even though the costs incurred will be the same, the time-value of money changes due to the discount-rate. As the fixed costs of a process are tied to the length of its useful life, those processes that do not fall on a period boundary require unique time-value multipliers in the objective function.

As noted above, Temoa allows the modeler to subdivide each year into a set of time slices, comprised of a season and a time of day. Unlike `time_future`, there is no restriction on what labels the modeler may assign to the `time_season` and `time_of_day` set elements.

5.2.1 A Word on Index Ordering

The ordering of the indices is consistent throughout the model to promote an intuitive "left-to-right" description of each parameter, variable, and constraint set. For example, Temoa's output commodity flow variable $FO_{r,p,s,d,i,t,v,o}$ may be described as "in region (r), in period (p) during season (s) at time of day (d), the flow of input commodity (i) to technology (t) of vintage (v) generates an output commodity flow (o) of $FO_{r,p,s,d,i,t,v,o}$." For any indexed parameter or variable within Temoa, our intent is to enable a mental model of a simple left-to-right, arrow-box-arrow mnemonic to describe the "input → process → output" flow of energy. And while not all variables, parameters, or constraints have 8 indices, the 8-index order mentioned here (r, p, s, d, i, t, v, o) is the canonical ordering. If you note any case where, for example, d comes before s, that is an oversight. In general, if there is an index ordering that does not follow this rubric, we view that as a bug.

5.2.2 Deviations from Standard Mathematical Notation

Temoa deviates from standard mathematical notation and set understanding in two ways. The first is that Temoa places a restriction on the *time* set elements. Specifically, while most optimization programs treat set elements as arbitrary labels, Temoa assumes that all elements of the `time_existing` and `time_future` sets are integers. Further, these sets are assumed to be ordered, such that the minimum element is "naught". For example, if $P^f = \{2015, 2020, 2030\}$, then $P_0 = 2015$. In other words, the capital P with the naught subscript indicates the first element in the `time_future` set. We will explain the reason for this notation shortly.

The second set of deviations revolves around the use of the Theta superset (Θ). The Temoa code makes heavy use of sparse sets, for both correctness and efficient use of computational resources. For brevity, and to avoid discussion of implementation details, we do not enumerate their logical creation here. Instead, we rely on the readers general understanding of the context. For example, in the sparse creation of the constraints of the Demand constraint class

(explained in *Network Constraints* and *Anatomy of a Constraint*), we state simply that the constraint is instantiated "for all the $\{p, s, d, dem\}$ tuples in Θ_{demand}". This means that the constraint is only defined for the exact indices for which the modeler specified end-use demands via the Demand parameter in the input data file.

Summations also occur in a sparse manner. For example, let's take another look at the Capacity (5.1) Constraint:

$$(\text{CFP}_{r,s,d,t,v} \cdot \text{C2A}_{r,t} \cdot \text{SEG}_{s,d} \cdot \text{TLF}_{r,p,t,v}) \cdot \mathbf{CAP}_{r,t,v} = \sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{I,O} \mathbf{CUR}_{r,p,s,d,i,t,v,o}$$

$$\forall \{p, s, d, t, v\} \in \Theta_{\text{Capacity}}$$

It defines the Capacity variable for every valid combination of $\{p, v\}$, and includes the sum over all inputs and outputs of the FlowOut variable. A naive implementation of this equation might include nonsensical items in each summation, such as an input of vehicle miles traveled to an oil refinery or an output of sunlight from nuclear generating capacity. However, in this context, summing over the inputs and outputs (i and o) implicitly includes only the valid combinations of $\{p, s, d, i, t, v, o\}$.

5.3 Parameters

Table 2: List of Temoa parameters with which a modeler might interact. The asterisked (*) parameters specified at the end of the table are automatically derived by the model and are not user-specifiable.

Parameter	Temoa Name	Domain	Short Description
$\text{CC}_{r,p,t,v}$	CapacityCredit	$\mathbb{I}$	Process-specific capacity credit
$\text{CFT}_{r,s,d,t}$	CapacityFactorTech	$\mathbb{I}$	Technology-specific capacity factor
$\text{CFP}_{r,s,d,t,v}$	CapacityFactorProcess	$\mathbb{I}$	Process-specific capacity factor
$\text{C2A}_{r,t,v}$	CapacityToActivity	$\mathbb{R}_0^+$	Converts from capacity to activity units
$\text{CF}_{r,p,t,v}$	CostFixed	$\mathbb{R}$	Fixed operations & maintenance cost
$\text{CI}_{r,t,v}$	CostInvest	$\mathbb{R}$	Tech-specific investment cost
$\text{CV}_{r,p,t,v}$	CostVariable	$\mathbb{R}$	Variable operations & maintenance cost
$\text{DEM}_{r,p,c}$	Demand	$\mathbb{R}_0^+$	End-use demands, by period
$\text{DDD}_{p,s,d}$	DemandDefaultDistribution	$\mathbb{I}$	Default demand distribution
$\text{DSD}_{r,p,s,d,c}$	DemandSpecificDistribution	$\mathbb{I}$	Demand-specific distribution
$\text{DR}_{r,t,v}$	DiscountRate	$\mathbb{R}$	Tech-specific interest rate on investment
$\text{EFF}_{r,i,t,v,o}$	Efficiency	$\mathbb{R}_0^+$	Tech- and commodity-specific efficiency
$\text{EAC}_{r,i,t,v,o,e}$	EmissionActivity	$\mathbb{R}$	Tech-specific emissions rate
$\text{ELM}_{r,p,e}$	EmissionLimit	$\mathbb{R}_0^+$	Emissions limit by region and period
$\text{ECAP}_{r,t,v}$	ExistingCapacity	$\mathbb{R}_0^+$	Pre-existing capacity
GDR	GlobalDiscountRate	$\mathbb{R}$	Global rate used to calculate present cost
$\text{GRM}_{r,t}$	GrowthRateMax	$\mathbb{R}$	Global rate used to calculate present cost
$\text{GRS}_{r,t}$	GrowthRateSeed	$\mathbb{R}$	Global rate used to calculate present cost
$\text{LLP}_{r,t,v}$	LifetimeLoanProcess	$\mathbb{N}$	Tech- and vintage-specific loan term
$\text{LLT}_{r,t}$	LifetimeLoanTech	$\mathbb{N}$	Tech-specific loan term
$\text{LTP}_{r,t,v}$	LifetimeProcess	$\mathbb{N}$	Tech- and vintage-specific lifetime
$\text{LTT}_{r,t}$	LifetimeTech	$\mathbb{N}$	Tech-specific lifetime
$\text{LIT}_{r,t,e,t}$	LinkedTechs	text	Dummy techs used to convert CO2 emissions to physical commodity
$\text{MAA}_{r,p,t}$	MaxActivity	$\mathbb{R}_0^+$	Maximum tech-specific activity by region and period
$\text{MAC}_{r,p,t}$	MaxCapacity	$\mathbb{R}_0^+$	Maximum tech-specific capacity by period

continues on next page

Table 2 – continued from previous page

Parameter	Temoa Name	Domain	Short Description
MCS_t	MaxCapacitySum	$\mathbb{R}_0^+$	Maximum capacity for a technology group
$MAR_{r,t}$	MaxResource	$\mathbb{R}_0^+$	Maximum resource production by tech across time periods
$MIA_{r,p,t}$	MinActivity	$\mathbb{R}_0^+$	Minimum tech-specific activity by region and period
$MIC_{r,p,t}$	MinCapacity	$\mathbb{R}_0^+$	Minimum tech-specific capacity by period
MCS_t	MinCapacitySum	$\mathbb{R}_0^+$	Minimum capacity for a technology group
MGT_r	MinGenGroupTarget	$\mathbb{R}_0^+$	Target applied to techs in MinActivityGroup constraint
$MGW_{r,t}$	MinGenGroupWeight	$\mathbb{R}_0^+$	Weight applied to techs in MinActivityGroup constraint
MBY	MyopicBaseYear	$\mathbb{N}$	Objective function base year when running myopically
PRM_r	PlanningReserveMargin	$\mathbb{I}$	Margin used to ensure sufficient generating capacity
$RMD_{r,t}$	RampDown	$\mathbb{R}$	Rate at which generation techs can ramp output down
$RMU_{r,t}$	RampUp	$\mathbb{R}$	Rate at which generation techs can ramp output up
$RSC_{r.p,c}$	ResourceBound	$\mathbb{R}_0^+$	Maximum resource production by tech and period
$SD_{r,t}$	StorageDuration	$\mathbb{N}$	Storage duration per technology, specified in hours
$SEG_{s,d}$	SegFrac	$\mathbb{I}$	Fraction of year represented by each (s, d) tuple
SIF_t	StorageInitFrac	$\mathbb{I}$	Initial storage charge level expressed as fraction of full charge
$TIS_{r,i,t}$	TechInputSplit	$\mathbb{I}$	Technology input fuel ratio at time slice level
$TISA_{r,i,t}$	TechInputSplitAverage	$\mathbb{I}$	Average annual technology input fuel ratio
$TOS_{r,t,o}$	TechOutputSplit	$\mathbb{I}$	Technology output fuel ratio at time slice level
*$LA_{t,v}$	LoanAnnualize	$\mathbb{R}_0^+$	Loan amortization by tech and vintage; based on DR_t
*$MPL_{p,t,v}$	ModelProcessLife	$\mathbb{N}$	Smaller of remaining model horizon or process tech life
*$PLF_{r,p,t,v}$	ProcessLifeFrac	$\mathbb{I}$	Fraction of available process capacity by region and period
*LEN_p	PeriodLength	$\mathbb{N}$	Number of years in period p

5.3.1 Efficiency

$EFF_{r \in R, i \in C_p, t \in T, v \in V, o \in C_c}$

We present the efficiency (EFF) parameter first as it is one of the most critical model parameters. Beyond defining the conversion efficiency of each process, Temoa also utilizes the indices to understand the valid input $\rightarrow$ process $\rightarrow$ output paths for energy. For instance, if a modeler does not specify an efficiency for a 2020 vintage coal power plant, then Temoa will recognize any mention of a 2020 vintage coal power plant elsewhere as an error. Generally, if a process is not specified in the efficiency table,[2] Temoa assumes it is not a valid process and will provide the user a warning with pointed debugging information.

---

[2] The efficiency parameter is often referred to as the efficiency table, due to how it looks after even only a few entries in the Pyomo input "dot dat" file.

5.3.2 CapacityCredit

$CC_{r \in R, p \in P, t \in T, v \in V}$

The capacity credit represents the fraction of total installed capacity of a process that can be relied upon during the time slice in which peak electricity demand occurs. This parameter is used in the $ReserveMargin$ constraint.

5.3.3 CapacityFactorTech

$CFT_{r \in R, s \in S, d \in D, t \in T}$

Temoa indexes the `CapacityFactorTech` parameter by season, time-of-day, and technology.

5.3.4 CapacityFactorProcess

$CFP_{r \in R, s \in S, d \in D, t \in T, v \in V}$

In addition to *CapacityCredit*, there may be cases where different vintages of the same technology have different capacity factors. For example, newer vintages of wind turbines may have higher capacity factors. So, `CapacityFactorProcess` allows users to specify the capacity factor by season, time-of-day, technology, and vintage.

5.3.5 CapacityToActivity

$C2A_{r \in R, t \in T}$

Capacity and Activity are inherently two different units of measure. Capacity represents the maximum flow of energy per time ($\frac{energy}{time}$), while Activity is a measure of total energy actually produced. However, there are times when one needs to compare the two, and this parameter makes those comparisons more natural. For example, a capacity of 1 GW for one year works out to an activity of

$$1GW \cdot 8,760\frac{hr}{yr} \cdot 3,600\frac{sec}{hr} \cdot 10^{-6}\frac{P}{G} = 31.536\frac{PJ}{yr}$$

or

$$1GW \cdot 8,760\frac{hr}{yr} \cdot 10^{-3}\frac{T}{G} = 8.75TWh$$

When comparing one capacity to another, the comparison is easy, unit wise. However, when one *needs* to compare capacity and activity, how does one reconcile the units? One way to think about the utility of this parameter is in the context of the question: "How much activity would this capacity create, if used 100% of the time?"

5.3.6 CostFixed

$CF_{r \in R, p \in P, t \in T, v \in V}$

The `CostFixed` parameter specifies the fixed cost associated with any process. Fixed costs are those that must be paid, regardless of how much the process is utilized. For instance, if the model decides to build a nuclear power plant, even if it decides not utilize the plant, the model must pay the fixed costs. These costs are in addition to the capital cost, so once the capital is paid off, these costs are still incurred every year the process exists.

Temoa's default objective function assumes the modeler has specified this parameter in units of currency per unit capacity ($\frac{Dollars}{UnitCap}$).

5.3.7 CostInvest

$CI_{r \in R, t \in T, v \in P}$

The `CostInvest` parameter specifies the process-specific investment cost. Unlike the `CostFixed` and `CostVariable` parameters, `CostInvest` only applies to vintages of technologies within the model optimization horizon (P^o). Like `CostFixed`, `CostInvest` is specified in units of cost per unit of capacity and is only used in the default objective function ($\frac{Dollars}{UnitCap}$).

5.3.8 CostVariable

$CV_{r \in R, p \in P, t \in T, v \in V}$

The `CostVariable` parameter represents the cost of a process-specific unit of activity. Thus the incurred variable costs are proportional to the activity of the process.

5.3.9 Demand

$DEM_{r \in r, p \in P, c \in C^d}$

The `Demand` parameter allows the modeler to define the total end-use demand levels for all periods. In combination with the `Efficiency` parameter, this parameter is the most important because without it, the rest of model has no incentive to build anything. This parameter specifies the end-use demands that appear at the far right edge of the system diagram.

To specify the distribution of demand, look to the `DemandDefaultDistribution` (DDD) and `DemandSpecificDistribution` (DSD) parameters.

As a historical note, this parameter was at one time also indexed by season and time of day, allowing modelers to specify exact demands for every time slice. However, while extremely flexible, this proved too tedious to maintain for any data set of appreciable size. Thus, we implemented the DDD and DSD parameters.

5.3.10 DemandDefaultDistribution

$DDD_{s \in S, d \in D}$

By default, Temoa assumes that end-use demands (*Demand*) are evenly distributed throughout a year. In other words, the Demand will be apportioned by the `SegFrac` parameter via:

$$\text{EndUseDemand}_{s,d,c} = SegFrac_{s,d} \cdot Demand_{p,c}$$

Temoa enables this default action by automatically setting DDD equivalent to `SegFrac` for all seasons and times of day. If a modeler would like a different default demand distribution, the indices and values of the DDD parameter must be specified. Like the *SegFrac* parameter, the sum of DDD must be 1.

5.3.11 DemandSpecificDistribution

$DSD_{r \in R, s \in S, d \in D, c \in C^d}$

If there is an end-use demand that varies over the course of a day or across seasons – for example, heating or cooling in the summer or winter – the modeler may specify the fraction of annual demand occurring in each time slice. Like *SegFrac* and *DemandDefaultDistribution*, the sum of DSD for each c must be 1. If the modeler does not define DSD for a season, time of day, and demand commodity, Temoa automatically populates this parameter according to DDD. It is this parameter that is actually multiplied by the `Demand` parameter in the Demand constraint.

5.3.12 DiscountRate

$DR_{r \in r, t \in T, v \in V}$

In addition to the `GlobalDiscountRate`, a modeler may also specify a technology-specific discount rate. If not specified, this rate defaults to 0.05.

5.3.13 EmissionActivity

$EAC_{e \in C_e, \{r,i,t,v,o\} \in \Theta_{\text{efficiency}}}$

Temoa currently has two methods for enabling a process to produce an output: the `Efficiency` parameter, and the `EmissionActivity` parameter. Where the `Efficiency` parameter defines the amount of output energy a process produces per unit of input, the `EmissionActivity` parameter allows for secondary outputs. As the name suggests, this parameter was originally intended to account for emissions per unit activity, but it more accurately describes *parallel* activity. It is restricted to emissions accounting (by the $e \in C^e$ set restriction).

5.3.14 EmissionLimit

$ELM_{r \in R, p \in P, e \in C^e}$

The `EmissionLimit` parameter ensures that Temoa finds a solution that fits within the modeler-specified limit of emission e in time period p.

5.3.15 ExistingCapacity

$ECAP_{r \in R, t \in T, v \in \mathrm{P}^e}$

The `ExistingCapacity` parameter defines the capacity installed prior to the beginning of `time_optimize`. Note that processes with existing capacity require all of the engineering-economic characteristics of a standard process, with the exception of an investment cost.

5.3.16 GlobalDiscountRate

GDR

The `GDR` parameter represents the global discount rate used to convert cash flows in future model time periods into a present value. The future value (FV) of a sum of currency is related to the net present value (NPV) via the formula:

$$\text{FV} = \text{NPV} \cdot (1 + GDR)^n$$

where n is in years. This parameter is only used in Temoa's objective function.

5.3.17 GrowthRateMax

$GRM_{r \in R, t \in T}$

The `GRM` parameter defines the maximum annual rate at which the capacity of a given technology can grow. Note that the growth rate is not defined by vintage, but rather across all vintages of a given technology.

5.3.18 GrowthRateSeed

$GRS_{r \in R, t \in T}$

The GRS parameter defines the maximum capacity of a given technology when first installed in a given time period. The growth rate is applied to this initial capacity seed in subsequent time periods.

5.3.19 LifetimeLoanProcess

$LLP_{r \in R, t \in T, v \in P}$

Temoa gives the modeler the ability to separate the loan lifetime from the useful life of a process. This parameter specifies the loan term associated with capital investment in a process, in years. If not specified, the model assigns the technology lifetime to the loan period in `temoa_initialize.py`.

5.3.20 LifetimeLoanTech

$LLT_{r \in R, t \in T}$

Same as the `LifetimeLoanProcess` but without the vintage index. If all vintages of a given technology are assumed to have the same loan term, then `LifetimeLoanTech` can be defined instead of `LifetimeLoanProcess`.

5.3.21 LifetimeProcess

$LTP_{r \in R, t \in T, v \in P}$

This parameter specifies the total useful life of a given process in years.

5.3.22 LifetimeTech

$LTT_{r \in R, t \in T}$

Similar to LifetimeProcess, this parameter specifies the total useful life of a given technology in years. If all vintages of a given technology have the same lifetime, then `LifeTimeTech` can be used instead of `LifeTimeProcess`.

5.3.23 LinkedTechs

$LIT_{r \in R, t \in T, e \in C^e, t \in T}$

In power-to-gas pathways, $CO2$ is an input to some processes, including synthetic natural gas production and liquid fuel production via Fischer-Tropsch. Within the model, $CO2$ must be converted from an emissions commodity to a physical commodity that can be included in the `Efficiency` table. The `LinkedTechs` parameter specifies the dummy technology used to convert an emissions commodity to a physical commodity. Note that the first `t` represents the primary upstream technology linked to the dummy linked technology, which is represented by the second `t` index.

5.3.24 MaxActivity

$MAA_{r \in R, p \in P, t \in T}$

The `MaxActivity` parameter is used to constrain the total activity (i.e., energy production) from a given technology in each model time period. Note that the total activity is constrained across all vintages of a technology. This parameter is used in the `MaxActivity_Constraint`.

5.3.25 MaxCapacity

$MAC_{r \in R, p \in P, t \in T}$

The `MaxCapacity` parameter represents an upper bound on the total installed capacity of a given technology in each model time period. Note that the total capacity is constrained across all vintages of a technology. This parameter is used in the `MaxCapacity_Constraint`.

5.3.26 MaxCapacitySum

$MCS_{t \in T}$

Similar to the `MaxCapacity` parameter, but represents an upper bound on the total installed capacity across all model time periods. In addition, this parameter specifies the upper bound on a group of technologies specified in the `tech_capacity_max` subset. This parameter is used in the `MaxCapacitySet_Constraint`.

5.3.27 MaxResource

$MAR_{r \in R, t \in T}$

The `MaxResource` parameter represents an upper bound on the cumulative amount of commodity that can be produced by region and technology over the model time horizon. This parameter is used in `MaxResource_Constraint`. Note that this parameter differs from `ResourceBound`, which is also indexed by model time period.

5.3.28 MinActivity

$MIA_{r \in R, p \in P, t \in T}$

The `MinActivity` parameter represents a lower bound on the total activity (i.e., energy production) of a given technology in each model time period. Note that the total activity is constrained across all vintages of a technology. This parameter is used in the `MinActivity_Constraint`.

5.3.29 MinCapacity

$MIC_{r \in R, p \in P, t \in T}$

The `MinCapacity` parameter represents a lower bound on the total installed capacity of a given technology in each model time period. Note that the total capacity is constrained across all vintages of a technology. This parameter is used in the `MinCapacity_Constraint`.

5.3.30 MinCapacitySum

$MCS_{t \in T}$

The `MinCapacitySum` parameter represents the minimum cumulative capacity associated with technologies belonging to `tech_group`. This parameter is used in the `MinActivityGroup_Constraint`.

5.3.31 MinGenGroupTarget

$MGT_{r \in R}$

The `MinGenGroupTarget` parameter is similar to `MinActivity`, but represents a minimum activity limit for a user-defined technology group (`tech_groups`) rather than a single technology. This parameter is used in the `MinActivityGroup_Constraint`.

5.3.32 MinGenGroupWeight

$MGW_{r \in R, t \in T}$

The `MinGenGroupWeight` parameter represents a weight that is applied to each technology within each `tech_group`, which determines the technology-specific activity shares that can count towards meeting the `MinActivityGroup_Constraint`.

5.3.33 MyopicBaseYear

MBY

Temoa is typically run in "perfect foresight" mode, where all decision variables in all time periods are solved simultaneously. However, it is also possible to solve the model myopically, whereby the model solves a subset of time periods in sequence. The `MyopicBaseYear` parameter specifies the base year to which all future costs are discounted.

5.3.34 PlanningReserveMargin

$PRM_{r \in R}$

The `PlanningReserveMargin` parameter specifies that capacity reserve margin in the electric sector by region. The capacity reserve margin represents the installed generating capacity - expressed as a share of peak load - that must be available to meet contingencies. Note that since electricity demand is often endogeous in Temoa databases, we calculate electricity production by time slice to estimate the peak electricity demand. This parameter is used in `ReserveMargin_Constraint`.

5.3.35 RampDown

$RMD_{r \in R, t \in T}$

To account for the limited ramping capability of some thermal generators, a ramp down rate can be specified via the `RampDown` parameter. The specified value represents the fraction of installed capacity that can be ramped down when moving from one time slice to the next. There is an equivalent `RampUp` parameter, to specify ramping limits in the upward direction. This parameter is used in the `RampDownDay_Constraint` and *RampDownSeason_Constraint*. The former constrains the downward ramp rate between time-of-day slices, and the latter constrains the downward ramp rate between the last time-of-day slice in a given season and the first time-of-day slice in the next season.

5.3.36 RampUp

$RMU_{r \in R, t \in T}$

To account for the limited ramping capability of some thermal generators, a ramp up rate can be specified via the `RampUp` parameter. The specified value represents the fraction of installed capacity that can be ramped up when moving from one time slice to the next. There is an equivalent `RampDown` parameter, to specify ramping limits in the downward direction. This parameter is used in the `RampUpDay_Constraint` and *RampUpSeason_Constraint*. The former constrains the upward ramp rate between time-of-day slices, and the latter constrains the upward ramp rate between the last time-of-day slice in a given season and the first time-of-day slice in the next season.

5.3.37 ResourceBound

$RSC_{r \in R, p \in P, c \in C_p}$

This parameter allows the modeler to specify commodity production limits per period. Note that a constraint in one period does not relate to any other periods. For instance, if the modeler specifies a limit in period 1 and does not specify a limit in period 2, then the model may use as much of that resource as it would like in period 2. This parameter is used in `ResourceExtraction_Constraint`. Note that the `MaxResource` parameter is similar, but constrains total cumulative resource consumption across all model time periods.

5.3.38 SegFrac

$SEG_{s \in S, d \in D}$

The `SegFrac` parameter specifies the fraction of the year represented by each combination of season and time of day. The sum of all combinations within `SegFrac` must be 1, representing 100% of a year.

5.3.39 StorageDuration

$SD_{r \in R, t \in T^S}$

The `StorageDuration` parameter represents the number of hours over which storage can discharge if it starts at full charge and produces maximum output until empty. The parameter value defaults to 4 hours if not specified by the user.

5.3.40 StorageInitFrac

$SI_{r \in R, t \in T^S, v \in P}$

The `StorageInitFrac` parameter determines the initial charge level associated with each storage technology. The value should be expressed as a fraction between 0 and 1. Note that this is an optional parameter and should only be used if the user wishes to set the initial charge rather than allowing the model to optimize it.

5.3.41 TechInputSplit

$TIS_{r \in R, p \in P, i \in C_p, t \in T}$

Some technologies have a single output but have multiple input fuels. The `TechInputSplit` parameter fixes the shares of commodity input to a specific technology in a given period. Note that this fixed share is maintained across all model time slices. This parameter is used in `TechInputSplit_Constraint`.

5.3.42 TechInputSplitAverage

$TISA_{r \in R, p \in P, i \in C_p, t \in T}$

The `TechInputSplitAverage` is similar to `TechInputSplit`, as they both fix input commodity shares to technologies with multiple inputs. However, `TechInputSplitAverage` only fixes the average shares at the annual level, allowing the shares at the time slice level to vary. This parameter is used in `TechInputSplitAverage_Constraint`.

5.3.43 TechOutputSplit

$TOS_{t \in T, o \in C_c}$

Some technologies have a single input fuel but have multiple outputs. The `TechOutputSplit` parameter fixes the shares of commodity input to a specific technology in a given period. Note that this fixed share is maintained across all model time slices. This parameter is used in `TechOutputSplit_Constraint`.

5.3.44 *LoanAnnualize

$LA_{r \in R, t \in T, v \in P}$

This is a model-calculated parameter based on the process-specific loan length (its indices are the same as the `LifetimeLoan` parameter), and process-specific discount rate (the `DiscountRate` parameter). It is calculated via the formula:

$$LA_{t,v} = \frac{DR_{r,t,v}}{1 - (1 + DR_{r,t,v})^{-LLN_{r,t,v}}}$$
$$\forall \{t, v\} \in \Theta_{\text{CostInvest}}$$

5.3.45 ModelProcessLife

$MPL_{r \in R, p \in P, t \in T, v \in P}$

The `ModelProcessLife` parameter is internally-derived by the model calcuated in `ParamModelProcessLife_rule` and which makes use of the `LifetimeProcess` parameter. For a given technology vintage in a given model time period, it returns the lesser of the period length and the remaining process lifetime. This parameter is used to sum the annual `fixed_costs` and `variable_costs` across all years within a given time period.

5.3.46 *PeriodLength

$LEN_{p \in P}$

Given that the modeler may specify arbitrary time period boundaries, this parameter specifies the number of years contained in each period. The final year is the largest element in `time_future` which is specifically not included in the list of periods in `time_optimize` (P^o). The length calculation for each period then exploits the fact that the `time` sets are ordered:

$$\text{LET boundaries} = \text{sorted}(P^f)$$
$$\text{LET I(p)} = \text{index of p in boundaries}$$
$$\therefore$$
$$LEN_p = \text{boundaries}[I(p) + 1] - p$$
$$\forall p \in P$$

The first line creates a sorted array of the period boundaries, called *boundaries*. The second line defines a function I that finds the index of period p in boundaries. The third line then defines the length of period p to be the number of years between period p and the next period. For example, if $P^f = \{2015, 2020, 2030, 2045\}$, then *boundaries* would be [`2015`, `2020`, `2030`, `2045`]. For 2020, I(2020) would return 2. Similarly, boundaries[3] = 2030. Then,

$$LEN_{2020} = \text{boundaries}[I(2020) + 1] - (2020)$$
$$= \text{boundaries}[2 + 1] - 2020$$
$$= \text{boundaries}[3] - 2020$$
$$= 2030 - 2020$$
$$= 10$$

Note that LEN is only defined for elements in P^o, and is specifically not defined for the final element in P^f.

5.3.47 *ProcessLifeFrac

$PLF_{r \in R, p \in P, t \in T, v \in P}$

The modeler may specify a useful lifetime of a process such that the process will be decommissioned part way through a period. Rather than attempt to delineate each year within that final period, Temoa averages the total output of the process over the entire period but limits the available capacity and output of the decommissioning process by the ratio of how long through the period the process is active. This parameter is that ratio, formally defined as:

$$PLF_{p,t,v} = \frac{v + LTP_{t,v} - p}{LEN_p}$$

$$\forall \{p, t, v\} \in \Theta_{\text{Activity by PTV}} \mid$$
$$v + LTP_{t,v} \notin P,$$
$$v + LTP_{t,v} \leq max(F),$$
$$p = max(P \mid p < v + LTP_{t,v})$$

Note that this parameter is defined over the same indices as `CostVariable` – the active periods for each process $\{p, t, v\}$. As an example, if a model has $P = \{2010, 2012, 2020, 2030\}$, and a process $\{t, v\} = \{car, 2010\}$ has a useful lifetime of 5 years, then this parameter would include only the first two activity indices for the process. Namely, $p \in \{2010, 2012\}$ as $\{p, t, v\} \in \{\{2010, car, 2010\}, \{2012, car, 2010\}\}$. The values would be $TLF_{2010,car,2010} = 1$, and $TLF_{2012,car,2010} = \frac{3}{8}$.

5.4 Variables

Table 3: Temoa's Main Variables

Variable	Temoa Name	Domain	Short Description
$FO_{r,p,s,d,i,t,v,o}$	V_FlowOut	$\mathbb{R}_0^+$	Commodity flow by time slice out of a tech based on a given input
$FOA_{r,p,s,d,i,t,v,o}$	V_FlowOutAnnual	$\mathbb{R}_0^+$	Annual commodity flow out of a tech based on a given input
$FIS_{r,p,s,d,i,t,v,o}$	V_FlowIn	$\mathbb{R}_0^+$	Commodity flow into a storage tech to produce a given output
$FLX_{r,p,s,d,i,t,v,o}$	V_Flex	$\mathbb{R}_0^+$	The portion of commodity production exceeding demand
$FLXA_{r,p,i,t,v,o}$	V_FlexAnnual	$\mathbb{R}_0^+$	The portion of commodity production from constant production techs exceeding demand
$CUR_{r,p,s,d,i,t,v,o}$	V_Curtailment	$\mathbb{R}_0^+$	Commodity flow out of a tech that is curtailed
$CAP_{r,t,v}$	V_Capacity	$\mathbb{R}_0^+$	Required tech capacity to support associated activity
$CAPAVL_{r,p,t}$	V_CapacityAvailable ByPeriodAndTech	$\mathbb{R}_0^+$	Derived variable representing the capacity of technology t available in period p
$SI_{r,t,v}$	V_StorageInit	$\mathbb{R}_0^+$	Initial charge level associated with storage techs
$SL_{r,p,s,d,t,v}$	V_StorageLevel	$\mathbb{R}_0^+$	Charge level each time slice associated with storage techs

5.4.1 V_FlowOut

$FO_{r,p,s,d,i,t,v,o}$

The most fundamental variable in the Temoa formulation is the `V_FlowOut` variable. It describes the commodity flow out of a process in a given time slice. To balance input and output flows in the `CommodityBalance_Constraint`, the commodity flow into a given process can be calculated as $\sum_{T,V,O} \mathbf{FO}_{p,s,d,c,t,v,o}/EFF_{c,t,v,o}$.

5.4.2 V_FlowOutAnnual

$FOA_{r,p,i,t,v,o}$

Similar to `V_FlowOut`, but used for technologies that are members of the `tech_annual` set, whose output does not vary across seasons and times-of-day. Eliminating the `s,d` indices for these technologies improves computational performance.

5.4.3 V_Flex

$FLX_{r,p,s,d,i,t,v,o}$

In some cases, the overproduction of a commodity may be required, such that supply exceeds the endogenous demand. Refineries represent a common example, where the share of different refined products are governed by TechOutput-Split, but total production is driven by a particular commodity. For example, gasoline production may be artificially constrained in order to ensure the appropriate balance for lower demand fuels such as propane or kerosene. Instead, we allow overproduction, i.e., production exceeding endogenous demand, for commodities produced by technologies belonging to the `tech_flex` set. In the example above, adding the refinery to the `tech_flex` set allows for the overproduction of propane and kerosene, allowing the model to fulfill the endogenous demand for gasoline. This flexible technology designation activates a slack variable ($\mathbf{FLX}_{r,p,s,d,i,t,v,c}$)representing the excess production in the `CommodityBalanceAnnual_Constraint`.

5.4.4 V_FlexAnnual

$FLXA_{r,p,i,t,v,o}$

Similar to `V_Flex`, but used for technologies that are members of the `tech_flex` set, whose output does not vary across seasons and times-of-day. Eliminating the `s,d` indices for these technologies improves computational performance.

5.4.5 V_Curtailment

$CUR_{r,p,s,d,i,t,v,o}$

The `V_Curtailment` variable allows for the overproduction and curtailment of technologies belonging to the `tech_curtailment` set. Renewables such as wind and solar are often placed in this set. While we used to simply formulate the `Capacity` and `CommodityBalance` constraints as inequalities that implicitly allowed for curtailment, this simpler approch does not work with renewable targets because the curtailed portion of the electricity production counts towards the target, and there is no way to distinguish it from the useful production. Including an explicit curtailment term addresses the issue.

5.4.6 V_FlowInStorage

$FIS_{r,p,s,d,i,t,v,o}$

Because the production and consumption associated with storage techs occur across different time slices, the comodity flow into a storage technologiy cannot be discerned from `V_FlowOut`. Thus an explicit $FlowIn$ variable is required for storage.

5.4.7 V_Capacity

$CAP_{r,t,v}$

The `V_Capacity` variable determines the required capacity of all processes across the user-defined system. It is indexed for each process (t,v), and Temoa constrains the capacity variable to be able to meet the total commodity flow out of that process in all time slices in which it is active (5.1).

5.4.8 V_CapacityAvailableByPeriodAndTech

$CAPAVL_{r,p,t}$

`CapacityAvailableByPeriodAndTech` is a convenience variable that is not strictly necessary, but used where the individual vintages of a technology are not warranted (e.g. in calculating the maximum or minimum total capacity allowed in a given time period).

5.4.9 V_StorageInit

$SI_{r,t,v}$

The `V_StorageInit` variable determines the initial storage charge level at the beginning of the first time slice within a given time period. Each vintage of each technology can have a different optimal initial value. Note that this value also determines the ending storage charge level at the end of the last time slice within each model time period.

5.4.10 V_StorageLevel

$SL_{r,p,s,d,t,v}$

The `V_StorageLevel` variable tracks the storage charge level across ordered time slices and is critical to ensure that storage charge and dispatch is constrained by the energy available in the storage units.

We explain the equations governing these variables the *Equations* section.

5.5 Equations

There are four main equations that govern the flow of energy through the model network. The `Demand_Constrant` (5.5) ensures that the supply meets demand in every time slice. For each process, the `Capacity_Constraint` (5.1) ensures that there is sufficient capacity to meet the optimal commodity flows across all time slices. Between processes, the `CommodityBalance_Constraint` (5.6) ensures that global commodity production across the energy system is sufficient to meet the endogenous demands for that commodity. Finally, the objective function (5.21) drives the model to minimize the system-wide cost of energy supply by optimizing the deployment and utilization of energy technologies across the system.

One additional point regarding the model formulation. Technologies that produce constant annual output can be placed in the `tech_annual` set. While not required, doing so improves computational performance by eliminating the season and time of day (`s,d`) indices associated with these technologies. In order to ensure the model functions correctly with these simplified technologies, slightly different formulations of the capacity and commodity balance constraints are required. See the `CommodityBalanceAnnual_Constraint` (5.7) and `CapacityAnnual_Constraint` (5.2) below for details.

The rest of this section defines each model constraint, with a rationale for existence. We use the implementation-specific names for the constraints to highlight the organization of the functions within the actual code. Note that the definitions below are pulled directly from the docstrings embedded in `temoa_rules.py`.

5.5.1 Constraints Defining Derived Decision Variables

These first four constraints define derived variables that are used within the model. The `Capacity_Constraint` and `CapacityAnnual_Constraint` are particularly important because they define the relationship between installed capacity and allowable commodity flow.

temoa_rules.**Capacity_Constraint**(M, r, p, s, d, t, v)

This constraint ensures that the capacity of a given process is sufficient to support its activity across all time periods and time slices. The calculation on the left hand side of the equality is the maximum amount of energy a process can produce in the timeslice (`s,d`). Note that the curtailment variable shown below only applies to technologies that are members of the curtailment set. Curtailment is necessary to track explicitly in scenarios that include a high renewable target. Without it, the model can generate more activity than is used to meet demand, and have all activity (including the portion curtailed) count towards the target. Tracking activity and curtailment separately prevents this possibility.

$$(\text{CFP}_{r,t,v} \cdot \text{C2A}_{r,t} \cdot \text{SEG}_{s,d} \cdot \text{PLF}_{r,p,t,v}) \cdot \mathbf{CAP}_{r,t,v} = \sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{I,O} \mathbf{CUR}_{r,p,s,d,i,t,v,o}$$

(5.1)

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{FO}}$$

temoa_rules.**CapacityAnnual_Constraint**(M, r, p, t, v)

Similar to Capacity_Constraint, but for technologies belonging to the `tech_annual` set. Technologies in the tech_annual set have constant output across different timeslices within a year, so we do not need to ensure that

installed capacity is sufficient across all timeslices, thus saving some computational effort. Instead, annual output is sufficient to calculate capacity.

$$(\text{CFP}_{r,t,v} \cdot \text{C2A}_{r,t} \cdot \text{PLF}_{r,p,t,v}) \cdot \mathbf{CAP}_{r,t,v} = \sum_{I,O} \mathbf{FOA}_{r,p,i,t \in T^a,v,o}$$

(5.2)

$$\forall \{r, p, t \in T^a, v\} \in \Theta_{\text{Activity}}$$

temoa_rules.**CapacityAvailableByPeriodAndTech_Constraint**(M, r, p, t)

The **CAPAVL** variable is nominally for reporting solution values, but is also used in the Max and Min constraint calculations. For any process with an end-of-life (EOL) on a period boundary, all of its capacity is available for use in all periods in which it is active (the process' PLF is 1). However, for any process with an EOL that falls between periods, Temoa makes the simplifying assumption that the available capacity from the expiring technology is available through the whole period in proportion to its remaining lifetime. For example, if a process expires 3 years into an 8-year model time period, then only $\frac{3}{8}$ of the installed capacity is available for use throughout the period.

$$\mathbf{CAPAVL}_{r,p,t} = \sum_{V} PLF_{r,p,t,v} \cdot \mathbf{CAP}_{r,t,v}$$

(5.3)

$$\forall p \in \mathbf{P}^o, r \in R, t \in T$$

temoa_rules.**ActivityByTech_Constraint**(M, t)

This constraint is utilized by the MGA objective function and defines the total activity of a technology over the planning horizon. The first version below applies to technologies with variable output at the timeslice level, and the second version applies to technologies with constant annual output in the `tech_annual` set.

$$\mathbf{ACT}_t = \sum_{R,P,S,D,I,V,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \ \forall t \notin T^a$$

$$\mathbf{ACT}_t = \sum_{R,P,I,V,O} \mathbf{FOA}_{r,p,i,t,v,o} \ \forall t \in T^a$$

(5.4)

5.5.2 Network Constraints

These three constraints define the core of the Temoa model; together, they define the algebraic energy system network.

temoa_rules.**Demand_Constraint**(M, r, p, s, d, dem)

The Demand constraint drives the model. This constraint ensures that supply at least meets the demand specified by the Demand parameter in all periods and slices, by ensuring that the sum of all the demand output commodity (c) generated by both commodity flow at the time slice level (**FO**) and the annual level (**FOA**) must meet the modeler-specified demand in each time slice.

$$\sum_{I,T-T^a,V} \mathbf{FO}_{r,p,s,d,i,t \notin T^a,v,dem} + SEG_{s,d} \cdot \sum_{I,T^a,V} \mathbf{FOA}_{r,p,i,t \in T^a,v,dem} = DEM_{r,p,dem} \cdot DSD_{r,s,d,dem}$$

(5.5)

Note that the validity of this constraint relies on the fact that the C^d set is distinct from both C^e and C^p. In other words, an end-use demand must only be an end-use demand. Note that if an output could satisfy both an end-use and internal system demand, then the output from **FO** and **FOA** would be double counted.

temoa_rules.**CommodityBalance_Constraint**(M, r, p, s, d, c)

Where the Demand constraint (5.5) ensures that end-use demands are met, the CommodityBalance constraint ensures that the endogenous system demands are met. This constraint requires the total production of a given commodity to equal the amount consumed, thus ensuring an energy balance at the system level. In this most general form of the constraint, the energy commodity being balanced has variable production at the time slice level. The energy commodity can then be consumed by three types of processes: storage technologies, non-storage technologies with output that varies at the time slice level, and non-storage technologies with constant annual output.

Separate expressions are required in order to account for the consumption of commodity c by downstream processes. For the commodity flow into storage technologies, we use $\mathbf{FI}_{r,p,s,d,i,t,v,c}$. Note that the FlowIn variable is defined only for storage technologies, and is required because storage technologies balance production and consumption across time slices rather than within a single time slice. For commodity flows into non-storage processes with time varying output, we use $\mathbf{FO}_{r,p,s,d,i,t,v,c}/EFF_{r,i,t,v,o}$. The division by $EFF_{r,c,t,v,o}$ is applied to the output flows that consume commodity c to determine input flows. Finally, we need to account for the consumption of commodity c by the processes in tech_annual. Since the commodity flow of these processes is on an annual basis, we use $SEG_{s,d}$ to calculate the consumption of commodity c in time-slice (s, d) from the annual flows. Formulating an expression for the production of commodity c is more straightforward, and is simply calculated by $\mathbf{FO}_{r,p,s,d,i,t,v,c}$.

In some cases, the overproduction of a commodity may be required, such that the supply exceeds the endogenous demand. Refineries represent a common example, where the share of different refined products are governed by TechOutputSplit, but total production is driven by a particular commodity like gasoline. Such a situation can result in the overproduction of other refined products, such as diesel or kerosene. In such cases, we need to track the excess production of these commodities. To do so, the technology producing the excess commodity should be added to the tech_flex set. This flexible technology designation will activate a slack variable ($\mathbf{FLX}_{r,p,s,d,i,t,v,c}$) representing the excess production in the CommodityBalanceAnnual_Constraint. Note that the tech_flex set is different from tech_curtailment set; the latter is technology- rather than commodity-focused and is used in the Capacity_Constraint to track output that is used to produce useful output and the amount curtailed, and to ensure that the installed capacity covers both.

This constraint also accounts for imports and exports between regions when solving multi-regional systems. The import (**FIM**) and export (**FEX**) variables are created on-the-fly by summing the **FO** variables over the appropriate import and export regions, respectively, which are defined in temoa_initialize.py by parsing the tech_exchange processes.

Finally, for commodities that are exclusively produced at a constant annual rate, the CommodityBalanceAnnual_Constraint is used, which is simplified and reduces computational burden.

production + imports = consumption + exports + excess

$$\sum_{I,T,V} \mathbf{FO}_{r,p,s,d,i,t,v,c} + \sum_{reg} \mathbf{FIM}_{r-reg,p,s,d,i,t,v,c} \; \forall reg \neq r$$

$$= \sum_{T^s,V,I} \mathbf{FIS}_{r,p,s,d,c,t,v,o}$$

$$+ \sum_{T-T^s,V,O} \mathbf{FO}_{r,p,s,d,c,t,v,o}/EFF_{r,c,t,v,o}$$

$$+ \; SEG_{s,d} \cdot \sum_{I,T^a,V} \mathbf{FOA}_{r,p,c,t\in T^a,v,o}/EFF_{r,c,t,v,o} \tag{5.6}$$

$$+ \sum_{reg} \mathbf{FEX}_{r-reg,p,s,d,c,t,v,o} \; \forall reg \neq r$$

$$+ \; \mathbf{FLX}_{r,p,s,d,i,t,v,c}$$

$$\forall\{r,p,s,d,c\} \in \Theta_{\text{CommodityBalance}}$$

temoa_rules.**CommodityBalanceAnnual_Constraint**(M, r, p, c)

Similar to the CommodityBalance_Constraint, but this version applies only to commodities produced at a constant annual rate. This version of the constraint improves computational performance for commodities that do not need to be balanced at the timeslice level.

While the commodity c can only be produced by technologies in the `tech_annual` set, it can be consumed by any technology in the $T - T^s$ set.

production + imports = consumption + exports + excess

$$\sum_{I,T,V} \mathbf{FOA}_{r,p,i,t\in T^a,v,c} + \sum_{reg} \mathbf{FIM}_{reg-r,p,i,t,v,c} \; \forall reg \neq r$$

$$= \sum_{S,D,T-T^s,V,O} \mathbf{FO}_{r,p,s,d,c,t,v,o}/EFF_{r,c,t,v,o}$$

$$+ \sum_{I,T^a,V,O} \mathbf{FOA}_{r,p,c,t\in T^a,v,o}/EFF_{r,c,t,v,o}$$

$$+ \sum_{reg} \mathbf{FEX}_{r-reg,p,c,t,v,o} \; \forall reg \neq r$$

$$+ \mathbf{FX}_{r,p,i,t,v,c}$$

(5.7)

$$\forall \{r, p, c\} \in \Theta_{\text{CommodityBalanceAnnual}}$$

5.5.3 Physical and Operational Constraints

These constraints fine-tune the model formulation to account for various physical and operational real-world phenomena.

temoa_rules.**BaseloadDiurnal_Constraint**(M, r, p, s, d, t, v)

Some electric generators cannot ramp output over a short period of time (e.g., hourly or daily). Temoa models this behavior by forcing technologies in the `tech_baseload` set to maintain a constant output across all times-of-day within the same season. Note that the output of a baseload process can vary between seasons.

Ideally, this constraint would not be necessary, and baseload processes would simply not have a d index. However, implementing the more efficient functionality is currently on the Temoa TODO list.

$$SEG_{s,D_0} \cdot \sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} = SEG_{s,d} \cdot \sum_{I,O} \mathbf{FO}_{r,p,s,D_0,i,t,v,o}$$

(5.8)

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{BaseloadDiurnal}}$$

temoa_rules.**DemandActivity_Constraint**($M, r, p, s, d, t, v, dem, s\_0, d\_0$)

For end-use demands, it is unreasonable to let the model arbitrarily shift the use of demand technologies across time slices. For instance, if household A buys a natural gas furnace while household B buys an electric furnace, then both units should be used throughout the year. Without this constraint, the model might choose to only use the electric furnace during the day, and the natural gas furnace during the night.

This constraint ensures that the ratio of a process activity to demand is constant for all time slices. Note that if a demand is not specified in a given time slice, or is zero, then this constraint will not be considered for that slice and demand. This is transparently handled by the Θ superset.

$$DEM_{r,p,s,d,dem} \cdot \sum_{I} \mathbf{FO}_{r,p,s_0,d_0,i,t\notin T^a,v,dem} = DEM_{r,p,s_0,d_0,dem} \cdot \sum_{I} \mathbf{FO}_{r,p,s,d,i,t\notin T^a,v,dem}$$

(5.9)

$$\forall \{r, p, s, d, t, v, dem, s_0, d_0\} \in \Theta_{\text{DemandActivity}}$$

Note that this constraint is only applied to the demand commodities with diurnal variations, and therefore the equation above only includes **FO** and not **FOA**

temoa_rules.**StorageEnergy_Constraint**(M, r, p, s, d, t, v)

This constraint tracks the storage charge level ($\mathbf{SL}_{r,p,s,d,t,v}$) assuming ordered time slices. The initial storage charge level is optimized for the first time slice in each period, and then the charge level is updated each time slice based on the amount of energy stored or discharged. At the end of the last time slice associated with each period, the charge level must equal the starting charge level. In the formulation below, note that **stored_energy** is an internal model decision variable.

First, the amount of stored energy in a given time slice is calculated as the difference between the amount of energy stored (first term) and the amount of energy dispatched (second term). Note that the storage device's roundtrip efficiency is applied on the input side:

$$\mathbf{stored\_energy} = \sum_{I,O} \mathbf{FIS}_{r,p,s,d,i,t,v,o} \cdot EFF_{r,i,t,v,o} - \sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \tag{5.10}$$

With **stored_energy** calculated, the storage charge level ($\mathbf{SL}_{r,p,s,d,t,v}$) is updated, but the update procedure varies based on the time slice within each time period. For the first season and time-of-day within a given period:

$$\mathbf{SL}_{r,p,s,d,t,v} = \mathbf{SI}_{r,t,v} + \mathbf{stored\_energy}$$

For the first time-of-day slice in any other season except the first:

$$\mathbf{SL}_{r,p,s,d,t,v} = \mathbf{SL}_{r,p,s_{prev},d_{last},t,v} + \mathbf{stored\_energy}$$

For the last season and time-of-day in the year, the ending storage charge level should be equal to the starting charge level:

$$\mathbf{SL}_{r,p,s,d,t,v} + \mathbf{stored\_energy} = \mathbf{SI}_{r,t,v}$$

For all other time slices not explicitly outlined above:

$$\mathbf{SL}_{r,p,s,d,t,v} = \mathbf{SL}_{r,p,s,d_{prev},t,v} + \mathbf{stored\_energy}$$

All equations below are sparsely indexed such that:

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{StorageEnergy}}$$

temoa_rules.**StorageEnergyUpperBound_Constraint**(M, r, p, s, d, t, v)

This constraint ensures that the amount of energy stored does not exceed the upper bound set by the energy capacity of the storage device, as calculated on the right-hand side.

Because the number and duration of time slices are user-defined, we need to adjust the storage duration, which is specified in hours. First, the hourly duration is divided by the number of hours in a year to obtain the duration as a fraction of the year. Since the $C2A$ parameter assumes the conversion of capacity to annual activity, we need to express the storage duration as fraction of a year. Then, $SEG_{s,d}$ summed over the time-of-day slices (d) multiplied by 365 days / yr yields the number of days per season. This step is necessary because conventional time sliced models use a single day to represent many days within a given season. Thus, it is necessary to scale the storage duration to account for the number of days in each season.

$$\mathbf{SL}_{r,p,s,d,t,v} \leq \mathbf{CAP}_{r,t,v} \cdot C2A_{r,t} \cdot \frac{SD_{r,t}}{8760 hrs/yr} \cdot \sum_{d} SEG_{s,d} \cdot 365 days/yr \tag{5.11}$$

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{StorageEnergyUpperBound}}$$

`temoa_rules.`**`StorageChargeRate_Constraint`**`(M, r, p, s, d, t, v)`

This constraint ensures that the charge rate of the storage unit is limited by the power capacity (typically GW) of the storage unit.

$$\sum_{I,O} \mathbf{FIS}_{r,p,s,d,i,t,v,o} \cdot EFF_{r,i,t,v,o} \leq \mathbf{CAP}_{r,t,v} \cdot C2A_{r,t} \cdot SEG_{s,d}$$

(5.12)

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{StorageChargeRate}}$$

`temoa_rules.`**`StorageDischargeRate_Constraint`**`(M, r, p, s, d, t, v)`

This constraint ensures that the discharge rate of the storage unit is limited by the power capacity (typically GW) of the storage unit.

$$\sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \leq \mathbf{CAP}_{r,t,v} \cdot C2A_{r,t} \cdot SEG_{s,d}$$

(5.13)

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{StorageDischargeRate}}$$

`temoa_rules.`**`StorageThroughput_Constraint`**`(M, r, p, s, d, t, v)`

It is not enough to only limit the charge and discharge rate separately. We also need to ensure that the maximum throughput (charge + discharge) does not exceed the capacity (typically GW) of the storage unit.

$$\sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{I,O} \mathbf{FIS}_{r,p,s,d,i,t,v,o} \cdot EFF_{r,i,t,v,o} \leq \mathbf{CAP}_{r,t,v} \cdot C2A_{r,t} \cdot SEG_{s,d}$$

(5.14)

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{StorageThroughput}}$$

`temoa_rules.`**`StorageInit_Constraint`**`(M, r, t, v)`

This constraint is used if the users wishes to force a specific initial storage charge level for certain storage technologies and vintages. In this case, the value of the decision variable $\mathbf{SI}_{r,t,v}$ is set by this constraint rather than being optimized. User-specified initial storage charge levels that are sufficiently different from the optimal $\mathbf{SI}_{r,t,v}$ could impact the cost-effectiveness of storage. For example, if the optimial initial charge level happens to be 50% of the full energy capacity, forced initial charge levels (specified by parameter $SIF_{r,t,v}$) equal to 10% or 90% of the full energy capacity could lead to more expensive solutions.

$$\mathbf{SI}_{r,t,v} \leq \ SIF_{r,t,v} \cdot \mathbf{CAP}_{r,t,v} \cdot C2A_{r,t} \cdot \frac{SD_{r,t}}{8760 hrs/yr} \cdot \sum_{d} SEG_{s_{first},d} \cdot 365 days/yr$$

(5.15)

$$\forall \{r, t, v\} \in \Theta_{\text{StorageInit}}$$

`temoa_rules.`**`RampUpDay_Constraint`**`(M, r, p, s, d, t, v)`

The ramp rate constraint is utilized to limit the rate of electricity generation increase and decrease between two adjacent time slices in order to account for physical limits associated with thermal power plants. Note that this constraint only applies to technologies with ramp capability, which is defined in the set T^m. We assume for simplicity the rate limits for both ramp up and down are equal and they do not vary with technology vintage. The ramp rate limits (r_t) for technology t should be expressed in percentage of its rated capacity.

Note that when d_{nd} is the last time-of-day, $d_{nd+1} \notin \mathbf{D}$, i.e., if one time slice is the last time-of-day in a season and the other time slice is the first time-of-day in the next season, the ramp rate limits between these two time slices can not be expressed by `RampUpDay`. Therefore, the ramp rate constraints between two adjacent seasons are represented in `RampUpSeason`.

In the `RampUpDay` and `RampUpSeason` constraints, we assume $\mathbf{S} = \{s_i, i = 1, 2, \cdots, ns\}$ and $\mathbf{D} = \{d_i, i = 1, 2, \cdots, nd\}$.

$$\frac{\sum_{I,O} \mathbf{FO}_{r,p,s,d_{i+1},i,t,v,o}}{SEG_{s,d_{i+1}} \cdot C2A_{r,t}} - \frac{\sum_{I,O} \mathbf{FO}_{r,p,s,d_i,i,t,v,o}}{SEG_{s,d_i} \cdot C2A_{r,t}} \leq r_t \cdot \mathbf{CAPAVL}_{r,p,t} \tag{5.16}$$

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{RampUpDay}}$$

temoa_rules.**RampDownDay_Constraint**(M, r, p, s, d, t, v)

Similar to the :code`RampUpDay` constraint, we use the `RampDownDay` constraint to limit ramp down rates between any two adjacent time slices.

$$\frac{\sum_{I,O} \mathbf{FO}_{r,p,s,d_{i+1},i,t,v,o}}{SEG_{s,d_{i+1}} \cdot C2A_{r,t}} - \frac{\sum_{I,O} \mathbf{FO}_{r,p,s,d_i,i,t,v,o}}{SEG_{s,d_i} \cdot C2A_{r,t}} \geq -r_t \cdot \mathbf{CAPAVL}_{r,p,t} \tag{5.17}$$

$$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{RampDownDay}}$$

temoa_rules.**RampUpSeason_Constraint**(M, r, p, s, t, v)

Note that d_1 and d_{nd} represent the first and last time-of-day, respectively.

$$\frac{\sum_{I,O} \mathbf{FO}_{r,p,s_{i+1},d_1,i,t,v,o}}{SEG_{s_{i+1},d_1} \cdot C2A_{r,t}} - \frac{\sum_{I,O} \mathbf{FO}_{r,p,s_i,d_{nd},i,t,v,o}}{SEG_{s_i,d_{nd}} \cdot C2A_{r,t}} \leq r_t \cdot \mathbf{CAPAVL}_{r,p,t} \tag{5.18}$$

$$\forall \{r, p, s, t, v\} \in \Theta_{\text{RampUpSeason}}$$

temoa_rules.**RampDownSeason_Constraint**(M, r, p, s, t, v)

Similar to the `RampUpSeason` constraint, we use the `RampDownSeason` constraint to limit ramp down rates between any two adjacent seasons.

$$\frac{\sum_{I,O} \mathbf{FO}_{r,p,s_{i+1},d_1,i,t,v,o}}{SEG_{s_{i+1},d_1} \cdot C2A_{r,t}} - \frac{\sum_{I,O} \mathbf{FO}_{r,p,s_i,d_{nd},i,t,v,o}}{SEG_{s_i,d_{nd}} \cdot C2A_{r,t}} \geq -r_t \cdot \mathbf{CAPAVL}_{r,p,t} \tag{5.19}$$

$$\forall \{r, p, s, t, v\} \in \Theta_{\text{RampDownSeason}}$$

temoa_rules.**ReserveMargin_Constraint**(M, r, p, s, d)

During each period p, the sum of the available capacity of all reserve technologies $\sum_{t \in T^{res}} \mathbf{CAPAVL}_{r,p,t}$, which are defined in the set $\mathbf{T}^{res}$, should exceed the peak load by PRM, the region-specific planning reserve margin. Note that the reserve margin is expressed in percentage of the peak load. Generally speaking, in a database we may not know the peak demand before running the model, therefore, we write this equation for all the time-slices defined in the database in each region.

$$\sum_{t \in T^{res}} CC_{t,r} \cdot \mathbf{CAPAVL}_{p,t} \cdot SEG_{s^*,d^*} \cdot C2A_{r,t} \geq \sum_{t \in T^{r,e}, V, I, O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \cdot (1 + PRM_r) \tag{5.20}$$

$$\forall \{r, p, s, d\} \in \Theta_{\text{ReserveMargin}}$$

5.5.4 Objective Function

temoa_rules.**TotalCost_rule**(M)

Using the `FlowOut` and `Capacity` variables, the Temoa objective function calculates the cost of energy supply, under the assumption that capital costs are paid through loans. This implementation sums up all the costs incurred, and is defined as $C_{tot} = C_{loans} + C_{fixed} + C_{variable}$. Each term on the right-hand side represents the

cost incurred over the model time horizon and discounted to the initial year in the horizon (P_0). The calculation of each term is given below.

$$C_{loans} = \sum_{r,t,v \in \Theta_{IC}} \left(\left[CI_{r,t,v} \cdot LA_{r,t,v} \cdot \frac{(1+GDR)^{P_0-v+1} \cdot (1-(1+GDR)^{-LLP_{r,t,v}})}{GDR} \right. \right.$$
$$\left. \left. \cdot \frac{1-(1+GDR)^{-LPA_{r,t,v}}}{1-(1+GDR)^{-LTP_{r,t,v}}} \right] \cdot \mathbf{CAP}_{r,t,v} \right) \tag{5.21}$$

Note that capital costs ($IC_{r,t,v}$) are handled in several steps. First, each capital cost is amortized using the loan rate (i.e., technology-specific discount rate) and loan period. Second, the annual stream of payments is converted into a lump sum using the global discount rate and loan period. Third, the new lump sum is amortized at the global discount rate and technology lifetime. Fourth, loan payments beyond the model time horizon are removed and the lump sum recalculated. The terms used in Steps 3-4 are $\frac{GDR}{1-(1+GDR)^{-LTP_{r,t,v}}} \cdot \frac{1-(1+GDR)^{-LPA_{t,v}}}{GDR}$. The product simplifies to $\frac{1-(1+GDR)^{-LPA_{r,t,v}}}{1-(1+GDR)^{-LTP_{r,t,v}}}$, where $LPA_{r,t,v}$ represents the active lifetime of process t in region r (r,t,v) before the end of the model horizon, and $LTP_{r,t,v}$ represents the full lifetime of a regional process (r,t,v). Fifth, the lump sum is discounted back to the beginning of the horizon (P_0) using the global discount rate. While an explicit salvage term is not included, this approach properly captures the capital costs incurred within the model time horizon, accounting for technology-specific loan rates and periods.

$$C_{fixed} = \sum_{r,p,t,v \in \Theta_{CF}} \left(\left[CF_{r,p,t,v} \cdot \frac{(1+GDR)^{P_0-p+1} \cdot (1-(1+GDR)^{-MPL_{r,t,v}})}{GDR} \right] \cdot \mathbf{CAP}_{r,t,v} \right) \tag{5.22}$$

$$C_{variable} =$$
$$\sum_{r,p,t,v \in \Theta_{CV}} \left(CV_{r,p,t,v} \cdot \frac{(1+GDR)^{P_0-p+1} \cdot (1-(1+GDR)^{-MPL_{r,p,t,v}})}{GDR} \cdot \sum_{S,D,I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \right)$$
$$+ \sum_{r,p,t \notin T^a, v \in \Theta_{VC}} \left(CV_{r,p,t,v} \cdot \frac{(1+GDR)^{P_0-p+1} \cdot (1-(1+GDR)^{-MPL_{r,p,t,v}})}{GDR} \cdot \sum_{I,O} \mathbf{FOA}_{r,p,i,t \in T^a,v,o} \right)$$
$$\tag{5.23}$$

5.5.5 User-Specific Constraints

The constraints provided in this section are not required for proper system operation, but allow the modeler some further degree of system specification.

temoa_rules.**ExistingCapacity_Constraint**(M, r, t, v)

    Temoa treats existing capacity installed prior to the beginning of the model's optimization horizon as regular processes that require the same parameter specification as do new vintage technologies, except for the `CostInvest` parameter. This constraint sets the capacity of processes for model periods that exist prior to the optimization horizon to user-specified values.

$$\mathbf{CAP}_{r,t,v} = ECAP_{r,t,v}$$
$$\forall \{r,t,v\} \in \Theta_{\text{ExistingCapacity}} \tag{5.24}$$

temoa_rules.**EmissionLimit_Constraint**(M, r, p, e)

    A modeler can track emissions through use of the `commodity_emissions` set and `EmissionActivity` parameter. The EAC parameter is analogous to the efficiency table, tying emissions to a unit of activity. The EmissionLimit constraint allows the modeler to assign an upper bound per period to each emission commodity. Note that this constraint sums emissions from technologies with output varying at the time slice and those with

constant annual output in separate terms.

$$\sum_{S,D,I,T,V,O|r,e,i,t,v,o \in EAC} (EAC_{r,e,i,t,v,o} \cdot \mathbf{FO}_{r,p,s,d,i,t,v,o})$$

$$+ \sum_{I,T,V,O|r,e,i,t \in T^a,v,o \in EAC} (EAC_{r,e,i,t,v,o} \cdot \mathbf{FOA}_{r,p,i,t \in T^a,v,o}) \leq ELM_{r,p,e} \quad (5.25)$$

$$\forall \{r,p,e\} \in \Theta_{\text{EmissionLimit}}$$

temoa_rules.**GrowthRateConstraint_rule**(M, p, r, t)

This constraint sets an upper bound growth rate on technology-specific capacity.

$$CAPAVL_{r,p_i,t} \leq GRM \cdot CAPAVL_{r,p_{i-1},t} + GRS$$

$$(5.26)$$

$$\forall \{r,p,t\} \in \Theta_{\text{GrowthRate}}$$

where GRM is the maximum growth rate, and should be specified as $(1+r)$ and GRS is the growth rate seed, which has units of capacity. Without the seed, any technology with zero capacity in the first time period would be restricted to zero capacity for the remainder of the time horizon.

temoa_rules.**MaxActivity_Constraint**(M, r, p, t)

The MaxActivity sets an upper bound on the activity from a specific technology. Note that the indices for these constraints are region, period and tech, not tech and vintage. The first version of the constraint pertains to technologies with variable output at the time slice level, and the second version pertains to technologies with constant annual output belonging to the `tech_annual` set.

$$\sum_{S,D,I,V,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \leq MAA_{r,p,t}$$

$$\forall \{r,p,t\} \in \Theta_{\text{MaxActivity}}$$

$$\sum_{I,V,O} \mathbf{FOA}_{r,p,i,t \in T^a,v,o} \leq MAA_{r,p,t} \quad (5.27)$$

$$\forall \{r,p,t \in T^a\} \in \Theta_{\text{MaxActivity}}$$

temoa_rules.**MinActivity_Constraint**(M, r, p, t)

The MinActivity sets a lower bound on the activity from a specific technology. Note that the indices for these constraints are region, period and tech, not tech and vintage. The first version of the constraint pertains to technologies with variable output at the time slice level, and the second version pertains to technologies with constant annual output belonging to the `tech_annual` set.

$$\sum_{S,D,I,V,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \geq MIA_{r,p,t}$$

$$\forall \{r,p,t\} \in \Theta_{\text{MinActivity}}$$

$$\sum_{I,V,O} \mathbf{FOA}_{r,p,i,t,v,o} \geq MIA_{r,p,t} \quad (5.28)$$

$$\forall \{r,p,t \in T^a\} \in \Theta_{\text{MinActivity}}$$

temoa_rules.**MinActivityGroup_Constraint**(M, p, g)

The MinActivityGroup constraint sets a minimum activity limit for a user-defined technology group. Each technology within each group is multiplied by a weighting function ($MGW_{r,t}$), which determines the technology

activity share that can count towards the constraint.

$$\sum_{S,D,I,T,V,O} \mathbf{FO}_{p,s,d,i,t,v,o} \cdot MGW_{t|t \notin T^a} + \sum_{I,T,V,O} \mathbf{FOA}_{p,i,t \in T^a,v,o} \cdot MGW_{t \in T^a} \geq MGT_{p,g}$$

(5.29)

$$\forall \{p,g\} \in \Theta_{\text{MinActivityGroup}}$$

where g represents the assigned technology group and MGT_r refers to the `MinGenGroupTarget` parameter.

temoa_rules.**MaxCapacity_Constraint**(M, r, p, t)

The MaxCapacity constraint sets a limit on the maximum available capacity of a given technology. Note that the indices for these constraints are region, period and tech, not tech and vintage.

$$\mathbf{CAPAVL}_{r,p,t} \leq MAC_{r,p,t}$$

$$\forall \{r,p,t\} \in \Theta_{\text{MaxCapacity}}$$

(5.30)

temoa_rules.**MaxCapacitySet_Constraint**(M, p)

Similar to the `MaxCapacity` constraint, but works on a group of technologies specified in the `tech_capacity_max` subset.

temoa_rules.**MinCapacity_Constraint**(M, r, p, t)

The MinCapacity constraint sets a limit on the minimum available capacity of a given technology. Note that the indices for these constraints are region, period and tech, not tech and vintage.

$$\mathbf{CAPAVL}_{r,p,t} \geq MIC_{r,p,t}$$

$$\forall \{r,p,t\} \in \Theta_{\text{MinCapacity}}$$

(5.31)

temoa_rules.**MinCapacitySet_Constraint**(M, p)

Similar to the `MinCapacity` constraint, but works on a group of technologies specified in the `tech_capacity_min` subset.

temoa_rules.**ResourceExtraction_Constraint**(M, reg, p, r)

The ResourceExtraction constraint allows a modeler to specify an annual limit on the amount of a particular resource Temoa may use in a period. The first version of the constraint pertains to technologies with variable output at the time slice level, and the second version pertains to technologies with constant annual output belonging to the `tech_annual` set.

$$\sum_{S,D,I,t \in T^r \& t \notin T^a,V} \mathbf{FO}_{r,p,s,d,i,t,v,c} \leq RSC_{r,p,c}$$

$$\forall \{r,p,c\} \in \Theta_{\text{ResourceExtraction}}$$

(5.32)

$$\sum_{I,t \in T^r \& t \in T^a,V} \mathbf{FOA}_{r,p,i,t,v,c} \leq RSC_{r,p,c}$$

$$\forall \{r,p,c\} \in \Theta_{\text{ResourceExtraction}}$$

temoa_rules.**TechInputSplit_Constraint**(M, r, p, s, d, i, t, v)

Allows users to specify fixed or minimum shares of commodity inputs to a process producing a single output. These shares can vary by model time period. See TechOutputSplit_Constraint for an analogous explanation. Under this constraint, only the technologies with variable output at the timeslice level (i.e., NOT in the `tech_annual` set) are considered.

temoa_rules.**TechOutputSplit_Constraint**(M, r, p, s, d, t, v, o)

Some processes take a single input and make multiple outputs, and the user would like to specify either a constant or time-varying ratio of outputs per unit input. The most canonical example is an oil refinery. Crude oil is used to produce many different refined products. In many cases, the modeler would like to specify a minimum share of each refined product produced by the refinery.

For example, a hypothetical (and highly simplified) refinery might have a crude oil input that produces 4 parts diesel, 3 parts gasoline, and 2 parts kerosene. The relative ratios to the output then are:

$$d = \tfrac{4}{9} \cdot \text{total output}, \qquad g = \tfrac{3}{9} \cdot \text{total output}, \qquad k = \tfrac{2}{9} \cdot \text{total output}$$

Note that it is possible to specify output shares that sum to less than unity. In such cases, the model optimizes the remaining share. In addition, it is possible to change the specified shares by model time period. Under this constraint, only the technologies with variable output at the timeslice level (i.e., NOT in the `tech_annual` set) are considered.

The constraint is formulated as follows:

$$\sum_{I, t \notin T^a} \mathbf{FO}_{r,p,s,d,i,t,v,o} \geq TOS_{r,p,t,o} \cdot \sum_{I,O, t \notin T^a} \mathbf{FO}_{r,p,s,d,i,t,v,o}$$

$$\forall \{r, p, s, d, t, v, o\} \in \Theta_{\text{TechOutputSplit}}$$

(5.33)

5.6 General Caveats

Temoa does not currently provide an easy avenue to track multiple concurrent energy flows through a process. Consider a cogeneration plant. Where a conventional power plant might simply emit excess heat as exhaust, a cogeneration plant harnesses some or all of that heat for heating purposes, either very close to the plant, or generally as hot water for district heating. Temoa's flow variables can track both flows through a process, but each flow will have its own efficiency from the Efficiency parameter. This implies that to produce 1 unit of electricity will require $\frac{1}{elceff}$ units of input. At the same time, to produce 1 unit of heat will require units of input energy, and to produce both output units of heat and energy, both flows must be active, and the desired activity will be double-counted by Temoa.

To model a parallel output device (c.f., a cogeneration plant), the modeler must currently set up the process with the `TechInputSplit` and `TechOutputSplit` parameters, appropriately adding each flow to the Efficiency parameter and accounting for the overall process efficiency through all flows.

THE TEMOA COMPUTATIONAL IMPLEMENTATION

We have implemented Temoa within an algebraic modeling environment (AME). AMEs provide both a convenient way to describe mathematical optimization models for a computational context, and allow for abstract model[7] formulations [Kallrath04]. In contrast to describing a model in a formal computer programming language like C or Java, AMEs generally have syntax that directly translates to standard mathematical notation. Consequently, models written in AMEs are more easily understood by a wider variety of researchers. Further, by allowing abstract formulations, a model written with an AME may be used with many different input data sets.

Three well-known and popular algebraic modeling environments are the General Algebraic Modeling System (GAMS) [BrookeRosenthal03], AMPL [FourerGayKernighan87], and GNU MathProg [Makhorin00]. All three environments provide concise syntax that closely resembles standard (paper) notation. We decided to implement Temoa within an AME called Python Optimization Modeling Objects (Pyomo).

Pyomo provides similar functionality to GAMS, AMPL, and MathProg, but is open source and written in the Python scripting language. This has two general consequences of which to be aware:

- Python is a scripting language; in general, scripts are an order of magnitude slower than an equivalent compiled program.

- Pyomo provides similar functionality, but because of its Python heritage, is **much** more verbose than GAMS, AMPL, or MathProg.

It is our view that the speed penalty of Python as compared to compiled languages is inconsequential in the face of other large resource bottle necks, so we omit any discussion of it as an issue. However, the "boiler-plate" code (verbosity) overhead requires some discussion. We discuss this in the *Anatomy of a Constraint*.

6.1 Anatomy of a Constraint

To help explain the Pyomo implementation, we discuss a single constraint in detail. Consider the `Demand` (5.5) constraint:

$$\sum_{I,T,V} \mathbf{FO}_{r,p,s,d,i,t,v,dem} + SEG_{s,d} \cdot \sum_{I,T^a,V} \mathbf{FOA}_{r,p,i,t,v,dem} \geq DEM_{r,p,dem} \cdot DSD_{r,s,d,dem}$$

$$\forall \{r, p, s, d, dem\} \in \Theta_{\text{Demand}}$$

Implementing this with Pyomo requires two pieces, and optionally a third:

1. a constraint definition (in `temoa_model.py`),

2. the constraint implementation (in `temoa_rules.py`), and

---

[7] In contrast to a 'concrete' model, an abstract algebraic formulation describes the general equations of the model, but requires modeler-specified input data before it can compute any results.

3. (optional) sparse constraint index creation (in `temoa_initialize.py`).

We discuss first a straightforward implementation of this constraint, that specifies the sets over which the constraint is defined. We will follow it with the actual implementation which utilizes a more computationally efficient but less transparent constraint index definition (the optional step 3).

A simple definition of this constraint is:

in `temoa_model.py`

```
1  M.DemandConstraint = Constraint(
2    M.regions, M.time_optimize, M.time_season, M.time_of_day, M.commodity_demand,
3    rule=Demand_Constraint
4  )
```

In line 1, 'M.DemandConstraint =' creates a place holder in the model object M, called 'DemandConstraint'. Like a variable, this is the name through which Pyomo will reference this class of constraints. `Constraint(...)` is a Pyomo-specific function that creates each individual constraint in the class. The first arguments (line 2) are the index sets of the constraint class. Line 2 is the Pyomo method of saying "for all" ($\forall$). Line 3 contains the final, mandatory argument (`rule=...`) that specifies the name of the implementation rule for the constraint, in this case `Demand_Constraint`. Pyomo will call this rule with each tuple in the Cartesian product of the index sets.

An associated implementation of this constraint based on the definition above is:

temoa_rules.py

...

```
1  def Demand_Constraint ( M, r, p, s, d, dem ):
2    if (r,p,s,d,dem) not in M.DemandSpecificDistribution.sparse_keys():  # If user did
   ↪not specify this Demand, tell
3      return Constraint.Skip              # Pyomo to ignore this constraint index.
4
5    supply = sum(
6      M.V_FlowOut[r, p, s, d, S_i, S_t, S_v, dem]
7      for S_t, S_v in M.commodityUStreamProcess[r, p, dem] if S_t not in M.tech_annual
8      for S_i in M.ProcessInputsByOutput[r, p, S_t, S_v, dem]
9    )
10
11   supply_annual = sum(
12     M.V_FlowOutAnnual[r, p, S_i, S_t, S_v, dem]
13     for S_t, S_v in M.commodityUStreamProcess[r, p, dem] if S_t in M.tech_annual
14     for S_i in M.ProcessInputsByOutput[r, p, S_t, S_v, dem]
15   ) * value( M.SegFrac[ s, d])
16
17   DemandConstraintErrorCheck(supply + supply_annual, r, p, s, d, dem)
18
19   expr = supply + supply_annual == M.Demand[r, p, dem] * M.
   ↪DemandSpecificDistribution[r, s, d, dem]
20   return expr
```

...

The Python boiler-plate code to create the rule is on line 1. It begins with `def`, followed by the rule name (matching the `rule=...` argument in the constraint definition in `temoa_model`), followed by the argument list. The argument

list will always start with the model (Temoa convention shortens this to just M) followed by local variable names in which to store the index set elements passed by Pyomo. Note that the ordering is the same as specified in the constraint definition. Thus the first item after M will be an item from region, the second from time_optimize, the third from time_season, fourth from time_of_day, and the fifth from commodity_demand. Though one could choose a, b, c, d, and e (or any naming scheme), we chose p, s, d, and dem as part of a *naming scheme* to aid in mnemonic understanding. Consequently, the rule signature (Line 1) is another place to look to discover what indices define a constraint.

Lines 2 and 3 are an indication that this constraint is implemented in a non-sparse manner. That is, Pyomo does not inherently know the valid indices for a given model parameter or equation. In temoa_model, the constraint definition listed five index sets, so Pyomo will naively call this function for every possible combination of tuple $\{r, p, s, d, dem\}$. However, as there may be slices for which a demand does not exist (e.g., the winter season might have no cooling demand), there is no need to create a constraint for any tuple involving 'winter' and 'cooling'. Indeed, an attempt to access a demand for which the modeler has not specified a value results in a Pyomo error, so it is necessary to ignore any tuple for which no Demand exists.

Lines 5 through 11 represent two *source-lines* that we split over several lines for clarity. These lines implement the summations of the demand commodity dem produced by demand technologies with both variable and constant output across the year, summed over all relevant technologies, vintages, and the inputs. The supply and supply_annual are local variables used in the expression (expr) shown below. Note that the sum is performed with sparse indices, which are returned from dictionaries created in temoa_initialize.py.

Lines 5 through 11 also showcase a very common idiom in Python: list-comprehension. List comprehension is a concise and efficient syntax to create lists. As opposed to building a list element-by-element with for-loops, list comprehension can convert many statements into a single operation. Consider a naive approach to calculating the supply:

```
to_sum = list()
for S_t in M.tech_all:
    for S_v in M.vintage_all:
        for S_i in ProcessInputsByOutput( p, S_t, S_v, dem ):
            to_sum.append( M.V_FlowOut[p, s, d, S_i, S_t, S_v, dem] )
supply = sum( to_sum )
```

This implementation creates an extra list (to_sum), then builds the list element by element with .append(), before finally calculating the summation. This means that the Python interpreter must iterate through the elements of the summation, not once, but twice.

A less naive approach would replace the .append() call with the += operator, reducing the number of iterations through the elements to one:

```
supply = 0
for S_t in M.tech_all:
    for S_v in M.vintage_all:
        for S_i in ProcessInputsByOutput( p, S_t, S_v, dem ):
            supply += M.V_FlowOut[p, s, d, S_i, S_t, S_v, dem]
```

Why is list comprehension necessary? Strictly speaking, it is not, especially in light of this last example, which may read more familiar to those comfortable with C, Fortran, or Java. However, due to quirks of both Python and Pyomo, list-comprehension is preferred both syntactically as "the Pythonic" way, and as the more efficient route for many list manipulations. (It also *may* seem slightly more familiar to those used to a more mainstream algebraic modeling language.)

With the correct model variables summed and stored in the supply and supply_annual variables, Line 17 calls a function defined in temoa_initialize.py that checks to make sure there is technology that can supply each demand commodity dem in each $\{r, p, s, d\}$.

If no process supplies the demand, then it quits computation immediately rather than completing a potentially lengthy

model generation and waiting for the solver to recognize the infeasibility of the model. Further, the function lists potential ways for the modeler to correct the problem. This is one of the benefits of Temoa: we've incorporated error handling in several places to try and capture the most common user errors. This capability is subtle, but in practice extremely useful while building and debugging a model.

Line 19 creates the actual inequality comparison. This line is superfluous, but we leave it in the code as a reminder that inequality operators (i.e. <= and >=) with a Pyomo object (like supply) generate a Pyomo *expression object*, not a boolean True or False as one might expect.[6] It is this expression object that must be returned to Pyomo, as on Line 20.

In the above implementation, the constraint is called for every tuple in the Cartesian product of the indices, and the constraint must then decide whether each tuple is valid. The below implementation differs from the one above because it only calls the constraint rule for the valid tuples within the Cartesian product, which is computationally more efficient than the simpler implementation above.

in `temoa_model.py` (actual implementation)

```
1  M.DemandConstraint_rpsdc = Set( dimen=5, rule=DemandConstraintIndices )
2  # ...
3  M.DemandConstraint = Constraint( M.DemandConstraint_rpsdc, rule=Demand_Constraint )
```

As discussed above, the DemandConstraint is only valid for certain $\{r, p, s, d, dem\}$ tuples. Since the modeler can specify the demand distribution per commodity (necessary to model demands like heating, that do not apply in all time slices), Temoa must ascertain the valid tuples. We have implemented this logic in the function `DemandConstraintIndices` in `temoa_initialize.py`. Thus, Line 1 tells Pyomo to instantiate `DemandConstraint_rpsdc` as a Set of 5-length tuples indices (`dimen=5`), and populate it with what Temoa's rule `DemandConstraintIndices` returns. We omit here an explanation of the implementation of the `DemandConstraintIndices` function, stating merely that it returns the exact indices over which the DemandConstraint must to be created. With the sparse set `DemandConstraint_rpsdc` created, we can now can use it in place of the five sets specified in the non-sparse implementation. Pyomo will now call the constraint implementation rule the minimum number of times.

On the choice of the `_rpsdc` suffix for the index set name, there is no Pyomo-enforced restriction. However, use of an index set in place of the non-sparse specification obfuscates over what indexes a constraint is defined. While it is not impossible to deduce, either from this documentation or from looking at the `DemandConstraintIndices` or `Demand_Constraint` implementations, the Temoa convention includes index set names that feature the one-character representation of each set dimension. In this case, the name `DemandConstraint_rpsdc` implies that this set has a dimensionality of 5, and (following the *naming scheme*) the first index of each tuple will be an element of `region`, the second an element of `time_optimize`, the third an element of `time_season`, fourth an element of `time_of_day`, and fifth a commodity. From the contextual information that this is the Demand constraint, one can assume that the `c` represents an element from `commodity_demand`.

---

[6] A word on *return* expressions in Pyomo: in most contexts a relational expression is evaluated instantly. However, in Pyomo, a relational expression returns an *expression* object. That is, '*M.aVar >= 5*' does not evaluate to a boolean *true* or *false*, and Pyomo will manipulate it into the final LP formulation.

6.2 A Word on Verbosity

Implementing this same constraint in AMPL, GAMS, or MathProg would require only a single source-line (in a single file). Using MathProg as an example, it might look like:

```
s.t. DemandConstraint{(p, s, d, dem) in sDemand_psd_dem} :
    sum{(p, s, d, Si, St, Sv, dem) in sFlowVar_psditvo}
      V_FlowOut[p, s, d, Si, St, Sv, dem]
  =
    pDemand[p, s, d, dem];
```

While the syntax is not a direct translation, the indices of the constraint (p, s, d, and dem) are clear, and by inference, so are the indices of summation (i, t, v) and operand (V_FlowOut). This one-line definition creates an inequality for each period, season, time of day, and demand, ensuring that total output meets each demand in each time slice – almost exactly as we have formulated the demand constraint (5.5). In contrast, Temoa's implementation in Pyomo takes 47 source-lines (the code discussed above does not include the function documentation). While some of the verbosity is inherent to working with a general purpose scripting language, and most of it is our formatting for clarity, the absolute minimum number of lines a Pyomo constraint can be is 2 lines, and that likely will be even less readable.

So why use Python and Pyomo if they are so verbose? In short, for four reasons:

- Temoa has the full power of Python, and has access to a rich ecosystem of tools (e.g. numpy, matplotlib) that are not as cleanly available to other AMLs. For instance, there is minimal capability in MathProg to error check a model before a solve, and providing interactive feedback like what Temoa's DemandConstraintErrorCheck function does is difficult, if not impossible. While a subtle addition, specific and directed error messages are an effective measure to reduce the learning curve for new modelers.

- Python has a vibrant community. Whereas mathematical optimization has a small community, its open-source segment even smaller, and the energy modeling segment significantly smaller than that, the Python community is huge, and encompasses many disciplines. This means that where a developer may struggle to find an answer, implementation, or workaround to a problem with a more standard AML, Python will likely enable a community-suggested solution.

- Powerful documentation tools. One of the available toolsets in the Python world is documentation generators that *dynamically* introspect Python code. While it is possible to inline and block comment with more traditional AMLs, the integration with Python that many documentation generators have is much more powerful. Temoa uses this capability to embed user-oriented documentation literally in the code, and almost every constraint has a block comment. Having both the documentation and implementation in one place helps reduce the mental friction and discrepancies often involved in maintaining multiple sources of model authority.

- AMLs are not as concise as thought.

This last point is somewhat esoteric, but consider the MathProg implementation of the Demand constraint in contrast with the last line of the Pyomo version:

```
expr = (supply = M.Demand[p, s, d, dem])
```

While the MathProg version indeed translates more directly to standard notation, consider that standard notation itself needs extensive surrounding text to explain the significance of an equation. *Why* does the equation compare the sum of a subset of FlowOut to Demand? In Temoa's implementation, a high-level understanding of what a constraint does requires only the last line of code: "Supply must meet demand."

6.3 File Structure

The Temoa model code is split into 7 main files:

- `temoa_model.py` - contains the overall model definition, defining the various sets, parameters, variables, and equations of the Temoa model. Peruse this file for a high-level overview of the model.

- `temoa_rules.py` - mainly contains the rule implementations. That is, this file implements the objective function, internal parameters, and constraint logic. Where `temoa_model` provides the high-level overview, this file provides the actual equation implementations.

- `temoa_initialize.py` - contains the code used to initialize the model, including sparse matrix indexing and checks on parameter and constraint specifications.

- `temoa_run.py` - contains the code required to execute the model when called with :code:'python' rather than :code:'pyomo solve'.

- `temoa_stochastic.py` - contains the PySP required alterations to the deterministic model for use in a stochastic model. Specifically, Temoa only needs one additional constraint class in order to partition the calculation of the objective function per period.

- `temoa_mga.py` - contains the functions used to execute the modeling-to- generate altenatives (MGA) algorithm. Use of MGA is specified through the config file.

- `pformat_results.py` - formats the results returned by the model; includes outputting results to the shell, storing them in a database, and if requested, calling 'DB_to_Excel.py' to create the Excel file outputs.

If you are working with a Temoa Git repository, these files are in the `temoa_model/` subdirectory.

6.4 The Bleeding Edge

The Temoa Project uses the Git source code management system, and the services of Github.com. If you are inclined to work with the bleeding edge of the Temoa Project code base, then take a look at the Temoa repository. To acquire a copy, make sure you have Git installed on your local machine, then execute this command to clone the repository:

```
$ git clone git://github.com/TemoaProject/temoa.git
Cloning into 'temoa'...
remote: Counting objects: 2386, done.
remote: Compressing objects: 100% (910/910), done.
remote: Total 2386 (delta 1552), reused 2280 (delta 1446)
Receiving objects: 100% (2386/2386), 2.79 MiB | 1.82 MiB/s, done.
Resolving deltas: 100% (1552/1552), done.
```

You will now have a new subdirectory called `temoa`, that contains the entire Temoa Project code and archive history. Note that Git is a *distributed* source code management tool. This means that by cloning the Temoa repository, you have your own copy to which you are welcome (and encouraged!) to alter and make commits to. It will not affect the source repository.

Though this is not a Git manual, we recognize that many readers of this manual may not be software developers, so we offer a few quick pointers to using Git effectively.

If you want to see the log of commits, use the command git log:

```
$ git log -1
commit b5bddea7312c34c5c44fe5cce2830cbf5b9f0f3b
Date:   Thu Jul 5 03:23:11 2012 -0400
```

```
    Update two APIs

    * I had updated the internal global variables to use the _psditvo
      naming scheme, and had forgotten to make the changes to _graphviz.py
    * Coopr also updated their API with the new .sparse_* methods.
```

You can also explore the various development branches in the repository:

```
$ ls
data_files  stochastic  temoa_model  create_archive.sh  README.txt

$ git branch -a
* energysystem
  remotes/origin/HEAD -> origin/energysystem
  remotes/origin/energysystem
  remotes/origin/exp_electric_load_duration_reorg
  remotes/origin/exp_electricity_sector
  remotes/origin/exp_energysystem_flow_based
  remotes/origin/exp_energysystem_match_markal
  remotes/origin/exp_energysystem_test_framework
  remotes/origin/misc_scripts
  remotes/origin/old_energysystem_coopr2
  remotes/origin/temoaproject.org

$ git checkout exp_energysystem_match_markal
Branch exp_energysystem_match_markal set up to track remote branch
exp_energysystem_match_markal from origin.
Switched to a new branch 'exp_energysystem_match_markal'

$ ls
temoa_model               create_archive.sh    utopia-markal-20.dat
compare_with_utopia-15.py  README.txt
compare_with_utopia-20.py  utopia-markal-15.dat
```

To view exactly what changes you have made since the most recent commit to the repository use the `diff` command to `git`:

```
$ git diff
diff --git a/temoa_model/temoa_lib.py b/temoa_model/temoa_lib.py
index 4ff9b30..0ba15b0 100644
--- a/temoa_model/temoa_lib.py
+++ b/temoa_model/temoa_lib.py
@@ -246,7 +246,7 @@ def InitializeProcessParameters ( M ):
            if l_vin in M.vintage_exist:
                if l_process not in l_exist_indices:
                    msg = ('Warning: %s has a specified Efficiency, but does␣
→not '
-                        'have any existing install base (ExistingCapacity)\n.')
+                        'have any existing install base (ExistingCapacity).\n')
                    SE.write( msg % str(l_process) )
                    continue
```

```
                    if 0 == M.ExistingCapacity[ l_process ]:
[ ... ]
```

For a crash course on git, here is a handy quick start guide.

TEMOA CODE STYLE GUIDE

It is an open question in programming circles whether code formatting actually matters. The Temoa Project developers believe that it does for these main reasons:

- Consistently-formatted code reduces the cognitive work required to understand the structure and intent of a code base. Specifically, we believe that before code is to be executed, it is to be understood by other humans. The fact that it makes the computer do something useful is a (happy) coincidence.

- Consistently-formatted code helps identify code smell.

- Consistently-formatted code helps one to spot code bugs and typos more easily.

Note, however, that this is a style *guide*, not a strict ruleset. There will also be corner cases to which a style guide does not apply, and in these cases, the judgment of what to do is left to the implementers and maintainers of the code base. To this end, the Python project has a well-written treatise in PEP 8:

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

1. When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.

2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up someone else's mess (in true XP style).

7.1 Indentation: Tabs and Spaces

The indentation of a section of code should always reflect the logical structure of the code. Python enforces this at a consistency level, but we make the provision here that **real tabs** (specifically **not spaces**) should be used at the beginning of lines. This allows the most flexibility across text editors and preferences for indentation width.

Spaces (and *not* tabs) should be used for mid-line spacing and alignment.

Many editors have functionality to highlight various whitespace characters.

7.2 End of Line Whitespace

Remove it. Many editors have plugins or builtin functionality that will take care of this automatically when the file is saved.

7.3 Maximum Line Length

(Similar to PEP 8) Limit all lines to a maximum of 80 characters.

Historically, 80 characters was the width (in monospace characters) that a terminal had to display output. With the advent of graphical user interfaces with variable font-sizes, this technological limit no longer exists. However, 80 characters remains an excellent metric of what constitutes a "long line." A long line in this sense is one that is not as transparent as to its intent as it could be. The 80-character width of code also represents a good "squint-test" metric. If a code-base has many lines longer than 80 characters, it may benefit from a refactoring.

Slightly adapted from PEP 8:

> The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation. Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is after the operator, not before it. Some examples:

```python
class Rectangle ( Blob ):

   def __init__ ( self, width, height,
                  color='black', emphasis=None, highlight=0 ):
      if ( width == 0 and height == 0 and
         color == 'red' and emphasis == 'strong' or
         highlight > 100 ):
         raise ValueError("sorry, you lose")
      if width == 0 and height == 0 and (color == 'red' or
                                         emphasis is None):
         raise ValueError("I don't think so -- values are {}, {}".format(
                        (width, height) ))
      Blob.__init__( self, width, height,
                     color, emphasis, highlight )
```

7.4 Blank Lines

- Separate logical sections within a single function with a single blank line.
- Separate function and method definitions with two blank lines.
- Separate class definitions with three blank lines.

7.5 Encodings

Following *PEP 3120*, all code files should use UTF-8 encoding.

7.6 Punctuation and Spacing

Always put spaces after code punctuation, like equivalence tests, assignments, and index lookups.

```
a=b               # bad
a = b             # good

a==b              # bad
a == b            # good

a[b] = c          # bad
a[ b ] = c        # good

   # exception: if there is more than one index
a[ b, c ] = d  # acceptable, but not preferred
a[b, c] = d    # good, preferred

   # exception: if using a string literal, don't include a space:
a[ 'x' ] == d  # bad
a['x'] == d    # good
```

When defining a function or method, put a single space on either side of each parenthesis:

```
def someFunction(a, b, c):      # bad
   pass

def someFunction ( a, b, c ):   # good
   pass
```

7.7 Vertical Alignment

Where appropriate, vertically align sections of the code.

```
    # bad
M.someVariable = Var( M.someIndex, domain=NonNegativeIntegers )
M.otherVariable = Var( M.otherIndex, domain=NonNegativeReals )

    # good
M.someVariable  = Var( M.someIndex,  domain=NonNegativeIntegers )
M.otherVariable = Var( M.otherIndex, domain=NonNegativeReals )
```

7.8 Single, Double, and Triple Quotes

Python has four delimiters to mark a string literal in the code: `"`, `'`, `"""`, and `'''`. Use each as appropriate. One should rarely need to escape a quote within a string literal, because one can merely alternate use of the single, double or triple quotes:

```
a = "She said, \"Do not do that!\""   # bad
a = 'She said, "Do not do that!"'     # good

b = "She said, \"Don't do that!\""    # bad
b = 'She said, "Don\'t do that!"'     # bad
b = """She said, "Don't do that!\""""  # bad
b = '''She said, "Don't do that!"'''  # good
```

7.9 Naming Conventions

All constraints attached to a model should end with `Constraint`. Similarly, the function they use to define the constraint for each index should use the same prefix and `Constraint` suffix, but separate them with an underscore (e.g. `M.somenameConstraint = Constraint( ..., rule=somename_Constraint)`:

```
M.CapacityConstraint = Constraint( M.CapacityVar_tv, rule=Capacity_Constraint )
```

When providing the implementation for a constraint rule, use a consistent naming scheme between functions and constraint definitions. For instance, we have already chosen `M` to represent the Pyomo model instance, `t` to represent *technology*, and `v` to represent *vintage*:

```
def Capacity_Constraint ( M, t, v ):
   ...
```

The complete list we have already chosen:

- p to represent a period item from $time\_optimize$
- s to represent a season item from $time\_season$
- d to represent a time of day item from $time\_of\_day$
- i to represent an input to a process, an item from $commodity\_physical$
- t to represent a technology from $tech\_all$

- v to represent a vintage from $vintage\_all$

- o to represent an output of a process, an item from $commodity\_carrier$

Note also the order of presentation, even in this list. In order to reduce the number mental "question marks" one might have while discovering Temoa, we attempt to rigidly reference a mental model of "left to right". Just as the entire energy system that Temoa optimizes may be thought of as a left-to-right graph, so too are the individual processes. As mentioned above in *A Word on Index Ordering*:

> For any indexed parameter or variable within Temoa, our intent is to enable a mental model of a left-to-right arrow-box-arrow as a simple mnemonic to describe the "input → process → output" flow of energy. And while not all variables, parameters, or constraints have 7 indices, the 7-index order mentioned here (p, s, d, i, t, v, o) is the canonical ordering. If you note any case where, for example, d comes before s, that is an oversight.

7.10 In-line Implementation Conventions

Wherever possible, implement the algorithm in a way that is *pedagogically* sound or reads like an English sentence. Consider this snippet:

```
if ( a > 5 and a < 10 ):
    doSomething()
```

In English, one might translate this snippet as "If a is greater than 5 and less then 10, do something." However, a semantically stronger implementation might be:

```
if ( 5 < a and a < 10 ):
    doSomething()
```

This reads closer to the more familiar mathematical notation of `5 < a < 10` and translates to English as "If a is between 5 and 10, do something." The semantic meaning that `a` should be *between* 5 and 10 is more readily apparent from just the visual placement between 5 and 10, and is easier for the "next person" to understand (who may very well be you in six months!).

Consider the reverse case:

```
if ( a < 5 or a > 10 ):
    doSomething()
```

On the number line, this says that a must fall before 5 or beyond 10. But the intent might more easily be understood if altered as above:

```
if not ( 5 < a and a < 10 ):
    doSomething()
```

This last snippet now makes clear the core question that a should `not` fall between 5 and 10.

Consider another snippet:

```
acounter = scounter + 1
```

This method of increasing or incrementing a variable is one that many mathematicians-turned-programmers prefer, but is more prone to error. For example, is that an intentional use of `acounter` or `scounter`? Assuming as written that it's incorrect, a better paradigm uses the `+=` operator:

```
acounter += 1
```

This performs the same operation, but makes clear that the `acounter` variable is to be incremented by one, rather than be set to one greater than `scounter`.

The same argument can be made for the related operators:

```
>>> a, b, c = 10, 3, 2

>>> a += 5;  a       # same as a = a + 5
15
>>> a -= b;  a       # same as a = a - b
12
>>> a /= b;  a       # same as a = a / b
4
>>> a *= c;  a       # same as a = a * c
8
>>> a **= c; a       # same as a = a ** c
64
```

7.11 Miscellaneous Style Conventions

- (Same as PEP 8) Do not use spaces around the assignment operator (=) when used to indicate a default argument or keyword parameter:

```
def complex ( real, imag = 0.0 ):        # bad
    return magic(r = real, i = imag)     # bad

def complex ( real, imag=0.0 ):          # good
    return magic( r=real, i=imag )       # good
```

- (Same as PEP 8) Do not use spaces immediately before the open parenthesis that starts the argument list of a function call:

```
a = b.calc ()        # bad
a = b.calc ( c )     # bad
a = b.calc( c )      # good
```

- (Same as PEP 8) Do not use spaces immediately before the open bracket that starts an indexing or slicing:

```
a = b ['key']        # bad
a = b [a, b]         # bad
a = b['key']         # good
a = b[a, b]          # good
```

7.12 Patches and Commits to the Repository

In terms of code quality and maintaining a legible "audit trail," every patch should meet a basic standard of quality:

- Every commit to the repository must include an appropriate summary message about the accompanying code changes. Include enough context that one reading the patch need not also inspect the code to get a high-level understanding of the changes. For example, "Fixed broken algorithm" does not convey much information. A more appropriate and complete summary message might be:

```
Fixed broken storage algorithm

The previous implementation erroneously assumed that only the energy
flow out of a storage device mattered.  However, Temoa needs to know the
energy flow in to all devices so that it can appropriately calculate the
inter-process commodity balance.

License: GPLv2
```

If there is any external information that would be helpful, such as a bug report, include a "clickable" link to it, such that one reading the patch as via an email or online, can immediately view the external information.

Specifically, commit messages should follow the form:

```
A subject line of 50 characters or less
 [ an empty line ]
1. http://any.com/
2. http://relevant.org/some/path/
3. http://urls.edu/~some/other/path/
4. https://github.com/blog/926-shiny-new-commit-styles
5. https://help.github.com/articles/github-flavored-markdown
 [ another empty line ]
Any amount and format of text, such that it conforms to a line-width of
72 characters[4].  Bonus points for being aware of the Github Markdown
syntax[5].

License: GPLv2
```

- Ensure that each commit contains no more than one *logical* change to the code base. This is very important for later auditing. If you have not developed in a logical manner (like many of us don't), `git add -p` is a very helpful tool.

- If you are not a core maintainer of the project, all commits must also include a specific reference to the license under which you are giving your code to the project. Note that Temoa will not accept any patches that are not licensed under GPLv2. A line like this at the end of your commit will suffice:

```
... the last line of the commit message.

License: GPLv2
```

This indicates that you retain all rights to any intellectual property your (set of) commit(s) creates, but that you license it to the Temoa Project under the terms of the GNU Public License, version 2. If the Temoa Project incorporates your commit, then Temoa may not relicense your (set of) patch(es), other than to increase the version number of the GPL license. In short, the intellectual property remains yours, and the Temoa Project would be but a licensee using your code similarly under the terms of GPLv2.

Executing licensing in this manner – rather than requesting IP assignment – ensures that no one group of code

contributers may unilaterally change the license of Temoa, unless **all** contributers agree in writing in a publicly archived forum (such as the Temoa Forum).

- When you are ready to submit your (set of) patch(es) to the Temoa Project, we will utilize GitHub's Pull Request mechanism.

BIBLIOGRAPHY

[BrookeRosenthal03] Anthony Brooke and Richard E. Rosenthal. *GAMS*. GAMS Development, 2003.

[DeCarolisHunterSreepathi13] Joseph DeCarolis, Kevin Hunter, and Sarat Sreepathi. Modeling for Insight using tools for energy model optimization and analysis (Temoa). *Energy Economics*, 40:339–349, 2013.

[FourerGayKernighan87] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Mathematical Programming Language*. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1987.

[Kallrath04] Josef Kallrath. *Modeling Languages in Mathematical Optimization*. Volume 88. Springer, 2004.

[Makhorin00] Andrew Makhorin. Modeling Language GNU MathProg. *Relatório Técnico*, 2000.

INDEX