

Tools for Energy Model Optimization and Analysis (Temoa)

Release 2020-02-01

Joe DeCarolus, Kevin Hunter

Feb 01, 2020

CONTENTS

1	Preface	1
1.1	What is Temoa?	1
1.2	Why Temoa?	2
1.3	Temoa Origin and Pronunciation	2
1.4	Bug Reporting	3
2	Quick Start	5
2.1	Installing Software Elements	5
2.2	Obtaining Temoa	5
2.3	Running Temoa	6
3	Database Construction	9
4	Visualization	13
4.1	Network Diagrams	13
4.2	Output Graphs	16
5	The Math Behind Temoa	17
5.1	Conventions	18
5.2	Sets	19
5.3	Parameters	21
5.4	Variables	28
5.5	Constraints	28
5.6	General Caveats	35
6	The Temoa Computational Implementation	37
6.1	Anatomy of a Constraint	37
6.2	A Word on Verbosity	41
6.3	File Structure	42
6.4	The Bleeding Edge	42
7	Temoa Code Style Guide	45
7.1	Indentation: Tabs and Spaces	46
7.2	End of Line Whitespace	46
7.3	Maximum Line Length	46
7.4	Blank Lines	47
7.5	Encodings	47
7.6	Punctuation and Spacing	47
7.7	Vertical Alignment	47
7.8	Single, Double, and Triple Quotes	48
7.9	Naming Conventions	48

7.10 In-line Implementation Conventions	49
7.11 Miscellaneous Style Conventions	50
7.12 Patches and Commits to the Repository	50
Bibliography	53
Index	55

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 explains the mathematical underpinnings 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 Github¹
- 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 realistic method 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 temoproject.org). 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.”

TĒMOĀ *vt* to seek something / buscar algo, o inquirir de algún negocio. This contrasts with TEMŌHUA, the nonactive form of TEMŌ ‘to descend.’

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. Navigate to this directory and execute the following from the command line:

```
$ conda env create
```

Then activate the environment as follows:

```
$ source activate temoa-py3
```

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 ‘coinbc’ 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. Another option is to run CPLEX on the [NEOS](#) server.

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 repository](#), 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

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). Note that if you do not have access to a commercial solver, it may be faster run cplex on the NEOS server. To do so, simply specify cplex as the solver and uncomment the '-neos' flag.

```
$ 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]

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

From the definition of the Temoa model as “an algebraic network of linked processes,” a directed network graph is a natural visualization. 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.

The programmatic interaction with Graphviz is entirely text based. The input files created by Temoa for Graphviz 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.

To utilize graphviz, make sure it is installed on your local machine. Then navigate to the `db_io` folder, where the graphviz script and database files reside. 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 temoa_utopia.sqlite
```

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`).

```
$ python MakeGraphviz.py -i 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:

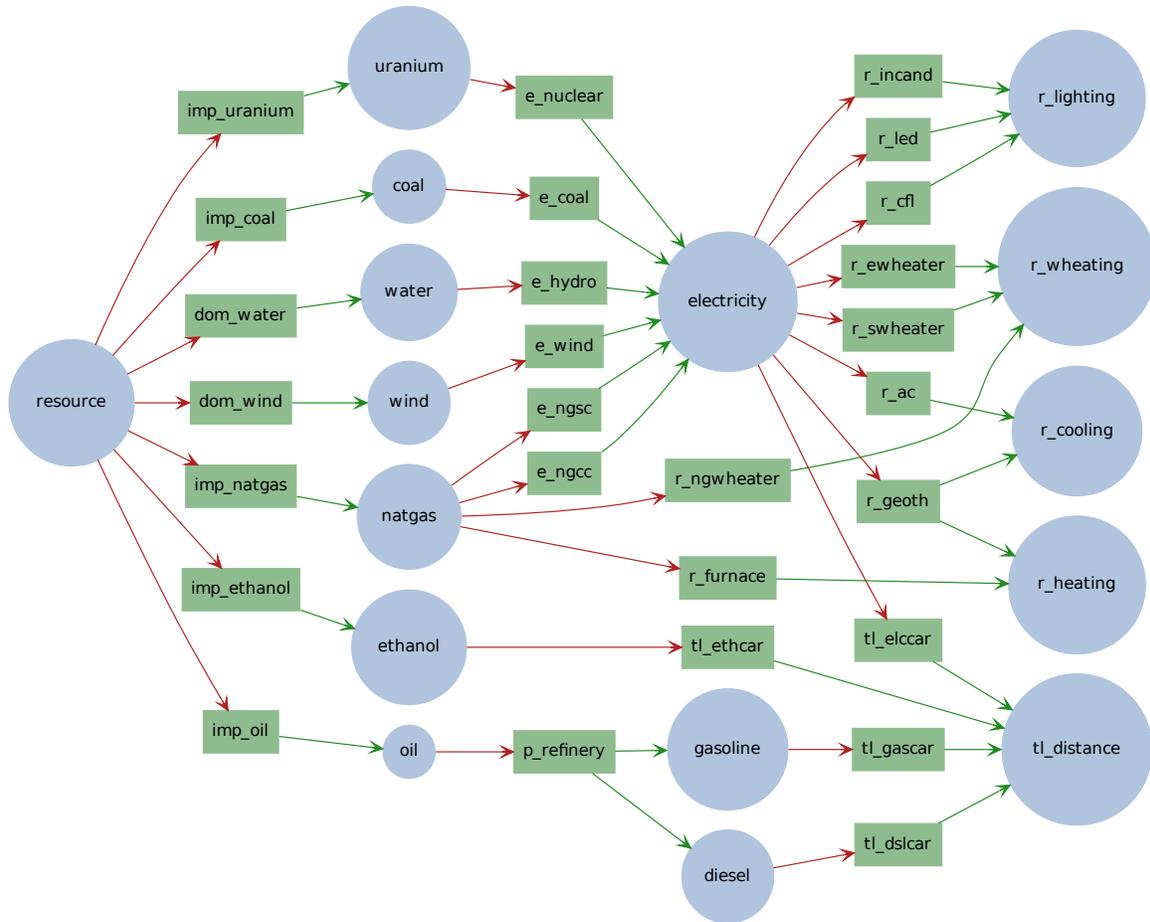


Fig. 1: This is a map of the simple ‘Utopia’ system, which we often use for testing purposes. The map shows the possible commodity flows through the system, providing a comprehensive overview of the system. Creating the simple system map is useful for debugging purposes in order to make sure that technologies are linked together properly via commodity flows.

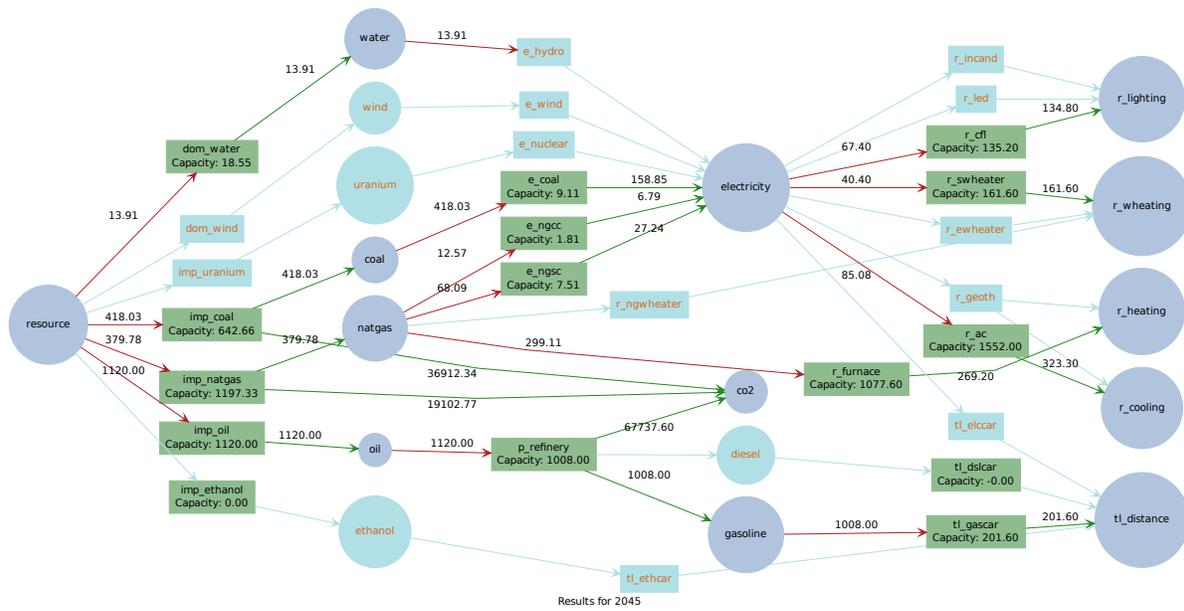


Fig. 2: This graph shows the optimal installed capacity and commodity flows from the ‘utopia’ test system in 2010.

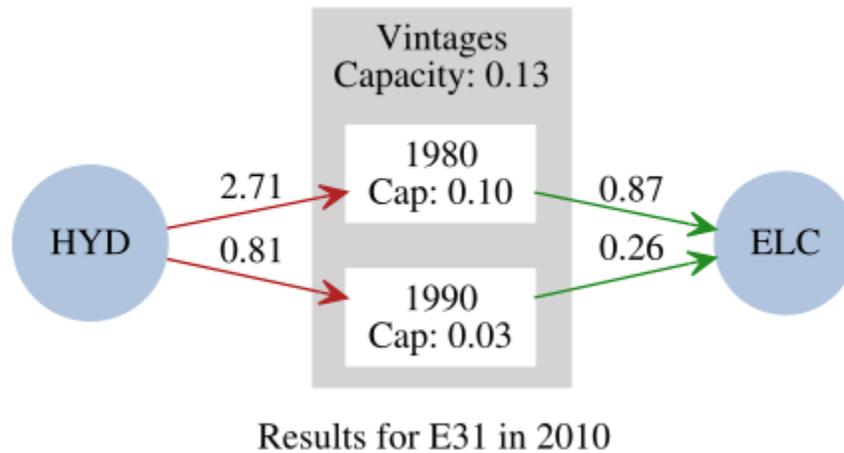


Fig. 3: In this case, the graph shows the commodity flow in and out of technology ‘E31’ in 2010, which is from the ‘test_run’ scenario drawn from the ‘temoa_utopia’ database.

```
$ python MakeGraphviz.py -i dbs/temoa_utopia.sqlite -s test_run -y 2010 -b E31
```

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 `db_io` folder and execute the following command:

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

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

```
$ python MakeOutputPlots.py -i dbs/temoa_utopia.sqlite -s test_run -p capacity -c_electric --super
```

Here is the result:

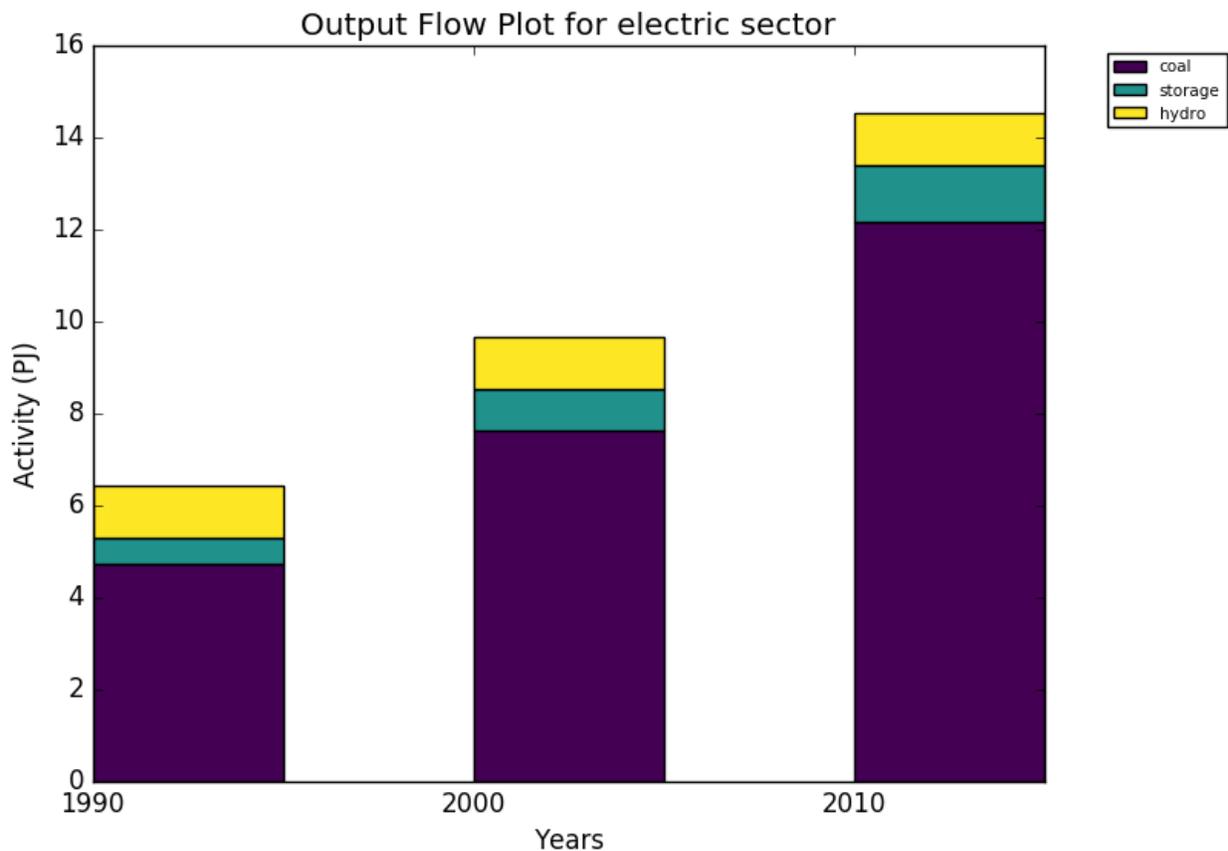


Fig. 4: This stacked bar plot represents the activity (i.e., output commodity flow) associated with each technology in the electric sector from the ‘test_run’ scenario drawn from the ‘temoa_utopia’ database. Because the `super` flag was specified, technologies are grouped together based on user-specified 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 – understood by the model as 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.

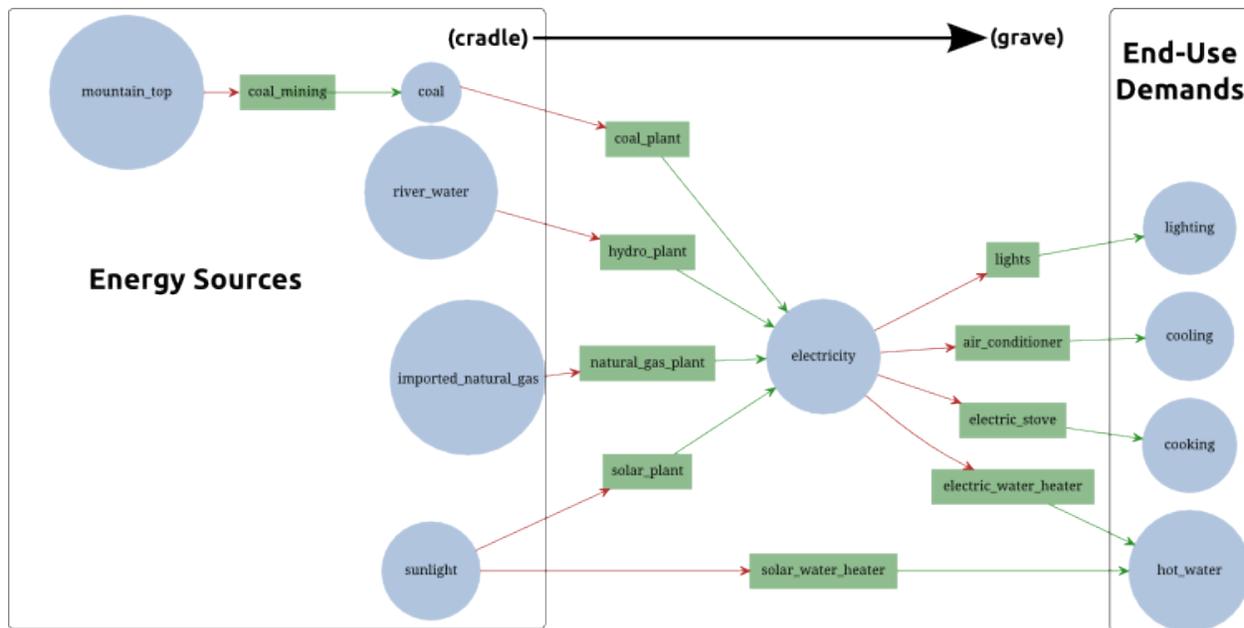
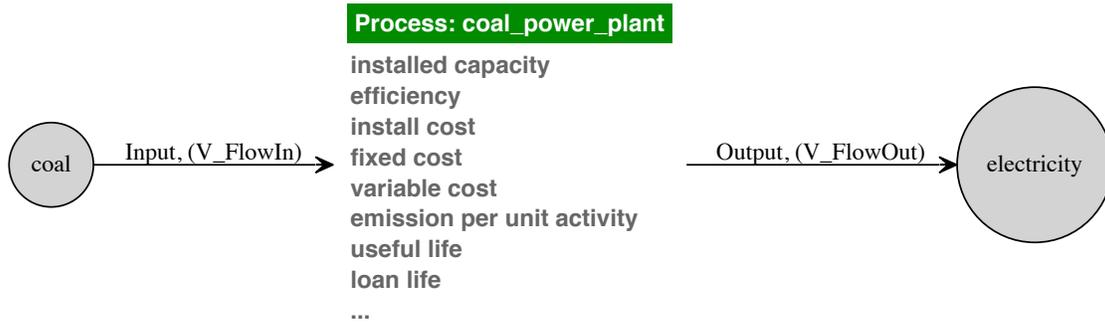


Fig. 1: A common visualization of energy system models is a directed network graph, with energy sources on the left and end-use demands on the right. The modeler must specify the specific end-use demands to be met, the technologies of the system (rectangles), and the inputs and outputs of each (red and green arrows). The circles represent distinct types of energy carriers.

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 an amalgam of sets and parameters, described in the next few sections. Temoa then translates these into variables and constraints that an optimizer may then solve.

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:

$$C_{variable} = \sum_{p,t,v \in \Theta_{VC}} (VC_{p,t,v} \cdot R_p \cdot \mathbf{ACT}_{t,v})$$

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:

$$C_{variable} = \sum_{p,t,v \in \Theta_{VC}} (\mathbf{ACT}_{t,v} \cdot VC_{p,t,v} \cdot R_p)$$

- 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 (5.2) defines Temoa’s mathematical understanding of a process capacity (**CAP**) in terms of that process’ activity (**ACT**):

$$(CF_{t,v} \cdot C2A_t \cdot SEG_{s,d} \cdot TLF_{p,t,v}) \cdot \mathbf{CAP}_{t,v} \geq \mathbf{ACT}_{p,s,d,t,v}$$

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

- We use the word ‘slice’ to refer to the tuple of season and time of day $\{s, d\}$. For example, “winter-night”.

- We use the word ‘process’ to refer to the tuple of technology and vintage ($\{t, v\}$), when knowing the vintage of a process is not pertinent to the context at hand.
 - In fact, in contrast to most other ESOMs, Temoa is “process centric.” This is a fairly large conceptual difference that we explain in detail in the rest of the documentation. However, it is a large enough point that we make it here for even the no-time quick-start modelers: think in terms of “processes” while modeling, not “technologies and start times”.
- 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

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 glyphs to denote sets, and small glyphs 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`. Note that regardless, the meanings are identical, with only minor interaction differences inherent to “implementation details.” [Table 1](#) lists all of the Temoa sets, with both notational schemes.

There are four basic set “groups” within Temoa: periods, annual “slices”, technology, and energy commodities. The technological 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 slice sets merit a slightly longer discussion.

Temoa’s conceptual model of *time* is broken up into three 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.
- **Seasonal** - Each year may have multiple seasons. For example, winter might demand more heating, while spring might demand more cooling and transportation.
- **Daily** - Within a season, a day might have various times of interest. For instance, the peak electrical load might occur midday in the summer, and a secondary peak might happen in the evening.

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. (This is part of Temoa’s answer to what most other efforts model as “residual capacity”.) 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 model. 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 a simplifying assumption that each period contains n copies of a single, representative year. Temoa optimizes just this characteristic

year, and only delineates each year within a period through a time-value of money calculation in the objective function. Figure 3.3 gives a graphical explanation of the annual delineation.

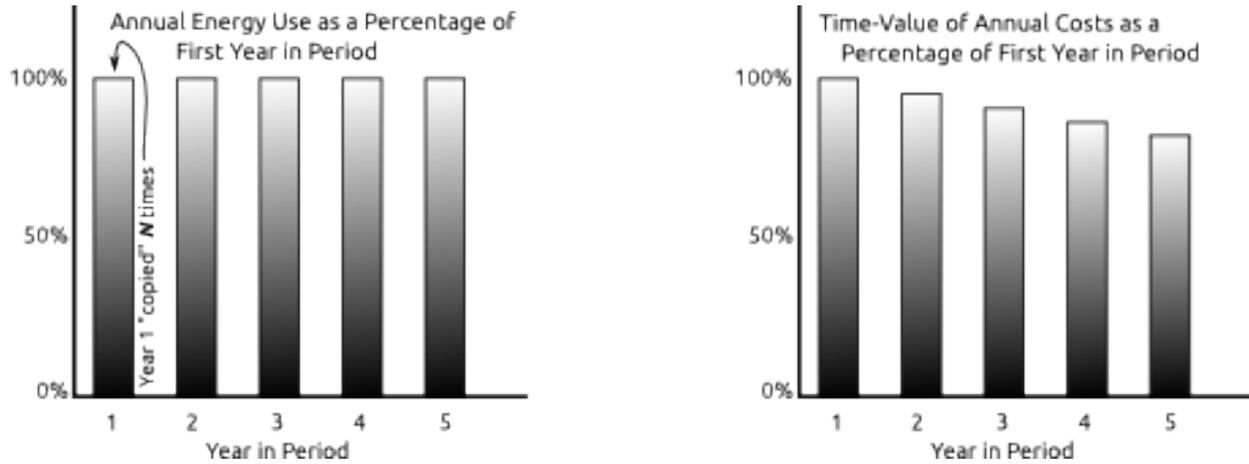


Fig. 2: The left graph is of energy, while the right graph is of the annual costs. In other words, the energy used in a period by a process is the same for all years (with exception for those processes that cease their useful life mid-period). However, even though the costs incurred will be the same, 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.

Many model-based analyses require sub-annual variations in demand as well. Temoa allows the modeler to subdivide years into slices, comprised of a season and a time of day (e.g. winter evening). Unlike the periods, 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 $FO_{p,s,d,i,t,v,o}$ may be described as “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 $FO_{p,s,d,i,t,v,o}$.” For any indexed parameter or variable within Temoa, our intent is to enable a mental model of a left-to-right arrow-box-arrow as a simple mnemonic to describe the “input \rightarrow process \rightarrow 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. 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 deviation 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 some “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.

Summations also occur in a sparse manner. Take equation (5.1) as an example (described in *Constraints Defining Derived Decision Variables*):

$$\text{ACT}_{p,s,d,t,v} = \sum_{I,O} \text{FO}_{p,s,d,i,t,v,o}$$

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

It defines the Activity variable for every valid combination of $\{p, s, d, t, v\}$ as 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 and an output of sunlight for a wind powered turbine. 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 1: List of Temoa parameters with which a modeler might interact. The asterisked (*) elements are automatically derived by the model and are not user-specifiable.

Parameter	Temoa Name	Domain	Short Description
CFT _{s,d,t}	CapacityFactorTech	\mathbb{I}	Technology-specific capacity factor
CFD _{s,d,t,v}	CapacityFactorProcess	\mathbb{I}	Process-specific capacity factor
C2A _{t,v}	Capacity2Activity	\mathbb{R}_0^+	Converts from capacity to activity units
FC _{p,t,v}	CostFixed	\mathbb{R}	Fixed operations & maintenance cost
IC _{t,v}	CostInvest	\mathbb{R}	Tech-specific investment cost
MC _{p,t,v}	CostVariable	\mathbb{R}	Variable operations & maintenance cost
DEM _{p,c}	Demand	\mathbb{R}_0^+	End-use demands, by period
DDD _{p,s,d}	DemandDefaultDistribution	\mathbb{I}	Default demand distribution
DSD _{p,s,d,c}	DemandSpecificDistribution	\mathbb{I}	Demand-specific distribution
DR _t	DiscountRate	\mathbb{R}	Tech-specific interest rate on investment
EFF _{i,t,v,o}	Efficiency	\mathbb{R}_0^+	Tech- and commodity-specific efficiency
EAC _{i,t,v,o,e}	EmissionsActivity	\mathbb{R}	Tech-specific emissions rate
ELM _{p,e}	EmissionsLimit	\mathbb{R}_0^+	Emissions limit by time period
ECAP _{t,v}	ExistingCapacity	\mathbb{R}_0^+	Pre-existing capacity
GDR	GlobalDiscountRate	\mathbb{R}	Global rate used to calculate present cost
GRM	GrowthRateMax	\mathbb{R}	Global rate used to calculate present cost
GRS	GrowthRateSeed	\mathbb{R}	Global rate used to calculate present cost
LLN _{t,v}	LifetimeLoanProcess	\mathbb{N}	Tech- and vintage-specific loan term
LTC _{p,t,v}	LifetimeTech	\mathbb{N}	Tech- and vintage-specific lifetime
MAX _{p,t}	MaxCapacity	\mathbb{R}_0^+	maximum tech-specific capacity by period
MIN _{p,t}	MinCapacity	\mathbb{R}_0^+	minimum tech-specific capacity by period
RSC _{p,c}	ResourceBound	\mathbb{R}_0^+	Upper bound on resource use
SEG _{s,d}	SegFrac	\mathbb{I}	Fraction of year represented by each (s, d) tuple
TIS _{i,t}	TechInputSplit	\mathbb{I}	Technology input fuel ratio
TOS _{t,o}	TechOutputSplit	\mathbb{I}	Technology output fuel ratio
*LA _{t,v}	LoanAnnualize	\mathbb{R}_0^+	Loan amortization by tech and vintage; based on DR_t
*MLL _{t,v}	ModelLoanLife	\mathbb{N}	Smaller of model horizon or process loan life

Continued on next page

Table 1 – continued from previous page

Parameter	Temoa Name	Domain	Short Description
*MPL _{p,t,v}	ModelTechLife	\mathbb{N}	Smaller of model horizon or process tech life
*LEN _p	PeriodLength	\mathbb{N}	Number of years in period <i>p</i>
*R _p	PeriodRate	\mathbb{R}	Converts future annual cost to discounted period cost
*TLF _{p,t,v}	TechLifetimeFrac	\mathbb{I}	Fraction of last time period that tech is active

5.3.1 Efficiency

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

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 → process → 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.

5.3.2 CapacityFactorTech

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

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

5.3.3 CapacityFactorProcess

$$CF_{s \in S, d \in D, t \in T, v \in V}$$

In addition to *CapacityFactorTech*, 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.

5.3.4 Capacity2Activity

$$C2A_{t \in T}$$

Capacity and Activity are inherently two different units of measure. Capacity represents the maximum flow of energy per time ($\frac{\text{energy}}{\text{time}}$), 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?”

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.

5.3.5 CostFixed

$$FC_{p \in P, t \in T, v \in V}$$

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 ($\frac{\text{Dollars}}{\text{UnitCap}}$).

5.3.6 CostInvest

$$IC_{t \in T, v \in P}$$

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 ($\frac{\text{Dollars}}{\text{UnitCap}}$).

5.3.7 CostVariable

$$MC_{p \in P, t \in T, v \in V}$$

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.

5.3.8 Demand

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

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.

5.3.9 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:

$$\text{EndUseDemand}_{s,d,c} = \text{SegFrac}_{s,d} \cdot \text{Demand}_{p,c}$$

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 modeler must specify any indices of the DDD parameter. Like the `SegFrac` parameter, the sum of DDD must be 1.

5.3.10 DemandSpecificDistribution

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

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.

5.3.11 DiscountRate

$$DR_{t \in T}$$

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

5.3.12 EmissionActivity

$$EAC_{e \in C^e, \{i, t, v, o\} \in \Theta_{\text{efficiency}}}$$

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).

5.3.13 EmissionLimit

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

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

5.3.14 ExistingCapacity

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

In contrast to some competing models, technologies in Temoa can have vintage-specific characteristics within the same period. Thus, Temoa treats existing technological capacity as processes, requiring all of the engineering characteristics of a standard process, with the exception of an investment cost.

5.3.15 GlobalDiscountRate

GDR

Because Temoa is a capacity expansion model, it must account for the time value of money. The future value (FV) of a sum of currency is related to the net present value (NPV) via the formula:

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

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

5.3.16 LifetimeLoan

$LLN_{t \in T, v \in P}$

Temoa gives the modeler the ability to separate the loan lifetime from the useful life of the technology. This parameter specifies the length of the loan associated with investing in a process, in years. If not specified, the default is 10 years.

5.3.17 LifetimeTech

$LTC_{p \in P, t \in T, v \in V}$

Similar to LifetimeLoan, this parameter specifies the total useful life of a given technology in years. If not specified, the default is 30 years.

5.3.18 MaxCapacity

$MAX_{p \in P, t \in T}$

The MaxCapacity parameter enables a modeler to ensure that a certain technology is constrained to an upper bound capacity. The constraint ensures that the max total capacity (summed across vintages) of a technology class is under this maximum. That is, all active vintages are constrained. This parameter is used only in the *maximum capacity constraint*.

5.3.19 MinCapacity

$MIN_{p \in P, t \in T}$

The MinCapacity parameter is analogous to the *MaxCapacity parameter*, except that it specifies the minimum capacity for which Temoa must ensure installation.

5.3.20 ResourceBound

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

This parameter allows the modeler to specify resources to constrain 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.

5.3.21 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.

5.3.22 TechInputSplit

$$SPL_{i \in C_p, t \in T}$$

Some technologies have a single output but have multiple input fuels. Some technologies require fixed shares of input. See the *TechOutputSplit constraint* for the implementation concept.

5.3.23 TechOutputSplit

$$SPL_{t \in T, o \in C_o}$$

Some technologies have a single input fuel but have multiple outputs. For the sake of modeling, certain technologies require fixed shares of output. For example, an oil refinery might have an input energy of crude oil, and the modeler wants to ensure that its output is 70% diesel and 30% gasoline. See the *TechOutputSplit constraint* for the implementation details.

5.3.24 *LoanAnnualize

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

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

$$LA_{t,v} = \frac{DR_{t,v}}{1 - (1 + DR_{t,v})^{-LLEN_{t,v}}}$$

$\forall \{t, v\} \in \Theta_{\text{CostInvest}}$

5.3.25 *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:

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

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,

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

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

5.3.26 *PeriodRate

$$R_{p \in P}$$

Temoa optimizes a single characteristic year within a period, and differentiates the n copies of that single year solely by the appropriate discount factor. Rather than calculating the same summation for every technology and vintage within a period, we calculate it once per period and lookup the sum as necessary during the objective function generation. The formula is the sum of discount factors corresponding to each year within a period:

$$R_p = \sum_{y=0}^{LEN_p} \frac{1}{(1 + GDR)^{(P_0 - p - y)}} \quad \forall p \in P$$

Note that this parameter is the implementation of the single “characteristic year” optimization per period concept discussed in the *Conventions* section.

5.3.27 *TechLifeFrac

$$TLF_{p \in P, t \in T, v \in V}$$

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 makes the choice to average the total output of the process over the entire period but limit 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:

$$\begin{aligned} TLF_{p,t,v} &= \frac{v + LTC_{t,v} - p}{LEN_p} \\ &\forall \{p, t, v\} \in \Theta_{\text{Activity by PTV}} \\ &v + LTC_{t,v} \notin P, \\ &v + LTC_{t,v} \leq \max(F), \\ &p = \max(P | p < v + LTC_{t,v}) \end{aligned}$$

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 $TLF_{2012,car,2010} = \frac{3}{8}$.

In combination with the `PeriodRate` parameter, this parameter is used to implement the “single characteristic year” simplification. Specifically, instead of trying to account for partial period decommissioning, Temoa assumes that processes can only produce `TechLifeFrac` of their installed capacity.

5.4 Variables

Table 2: Temoa’s Main Variables

Variable	Temoa Name	Domain	Short Description
$FI_{p,s,d,i,t,v,o}$	V_FlowIn	\mathbb{R}_0^+	Commodity flow into a tech to produce a given output
$FO_{p,s,d,i,t,v,o}$	V_FlowOut	\mathbb{R}_0^+	Commodity flow out of a tech based on a given input
$CUR_{p,s,d,i,t,v,o}$	V_Curtailment	\mathbb{R}_0^+	Commodity flow out of a tech that is curtailed
$ACT_{p,s,d,t,v}$	V_Activity	\mathbb{R}_0^+	Total tech commodity production in each (s, d) tuple
$CAP_{t,v}$	V_Capacity	\mathbb{R}_0^+	Required tech capacity to support associated activity
$CAPAVL_{p,t}$	V_CapacityAvailableByPeriodAndTech	\mathbb{R}_0^+	The capacity of technology t available in period p

The most fundamental variables in the Temoa formulation are *FlowIn* and *FlowOut*. They describe the commodity flows into and out of a process in a given time slice. They are related through the `ProcessBalance` constraint (5.5), which in essence, guarantees the conservation of energy for each process.

The Activity variable is defined as the sum over all inputs and outputs of a process in a given time slice (see equation (5.1)). At this time, one potential “gotcha” is that for a process with multiple inputs or outputs, there is no attempt to reconcile energy units: Temoa assumes all inputs are comparable, and has no understanding of units. The onus is on the modeler to ensure that all inputs and outputs have similar units.

The Capacity variable is used in the objective function as the amount of capacity of a process to build. It is indexed for each process, and Temoa constrains the Capacity variable to at least be able to meet the Activity of that process in all time slices in which it is active (5.2).

Finally, `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).

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

5.5 Constraints

There are 4 main equations that govern the flow of energy through the model network. The `DemandConstraint` ensures that the supply meets demand in every time slice. For each process, the `ProcessBalance` ensures at least as much energy enters a process as leaves it (conservation of energy at the process level). Between processes, the `CommodityBalance` ensures that at least as much of a commodity is generated as is demanded by other process inputs.

In combination, those three constraints ensure the flow of energy through the system. The final calculation, the objective function, is what puts a cost to the actions prescribed by the model.

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. They are listed roughly in order of importance. 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 two constraints elucidate the relationship among decision variables in the model. There is some overlap with the rest of the constraints, but these are unique enough to warrant special attention to a Temoa modeler.

`temoa_rules.Activity_Constraint` (M, p, s, d, t, v)

The Activity constraint defines the Activity convenience variable. The Activity variable is mainly used in the objective function to calculate the cost associated with use of a technology. This constraint sums the $\mathbf{FO}_{p,s,d,i,t,v,o}$ over all input and output commodities.

There is one caveat to keep in mind in regards to the Activity variable: if there is more than one output, there is currently no attempt by Temoa to convert to a common unit of measurement. For example, common measurements for heat include mass of steam at a given temperature, or total BTUs, while electricity is generally measured in a variant of watt-hours. Reconciling these units of measurement, as for example with a cogeneration plant, is currently left as an accounting exercise for the modeler.

$$\mathbf{ACT}_{p,s,d,t,v} = \sum_{I,O} \mathbf{FO}_{p,s,d,i,t,v,o} \quad (5.1)$$

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

`temoa_rules.Capacity_Constraint` (M, 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 $\langle s, d \rangle$. Note that the curtailment variable shown below only applies to technologies 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.

$$(\mathbf{CFP}_{t,v} \cdot \mathbf{C2A}_t \cdot \mathbf{SEG}_{s,d} \cdot \mathbf{TLF}_{p,t,v}) \cdot \mathbf{CAP}_{t,v} = \mathbf{ACT}_{p,s,d,t,v} + \sum_{I,O} \mathbf{CUR}_{p,s,d,i,t,v,o} \quad (5.2)$$

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

`temoa_rules.CapacityAvailableByPeriodAndTech_Constraint` (M, 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' TLF 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, but only as much percentage as its lifespan through the period. For example, if a process expires 3 years into an 8 year period, then only $\frac{3}{8}$ of the installed capacity is available for use throughout the period.

$$\mathbf{CAPAVL}_{p,t} = \sum_V \mathbf{TLF}_{p,t,v} \cdot \mathbf{CAP} \quad (5.3)$$

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

5.5.2 Network Constraints

These three constraints define the core of the Temoa model. Together, they create the algebraic network. The Demand constraint drives the “right side” of the energy system map, the ProcessBalance constraint ensures flow through a process, and the CommodityBalance constraint ensures flow between processes.

`temoa_rules.Demand_Constraint` (M, 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 **FO** must meet the modeler-specified demand, in each time slice.

$$\sum_{I,T,V} \mathbf{FO}_{p,s,d,i,t,v,dem} = DEM_{p,dem} \cdot DSD_{s,d,dem} \quad (5.4)$$

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

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** would be double counted.

`temoa_rules.ProcessBalance_Constraint` (M, p, s, d, i, t, v, o)

The ProcessBalance constraint is one of the most fundamental constraints in Temoa. It defines the basic relationship between the energy entering a process (**FI**) and the energy leaving a process (**FO**). This constraint sets the `FlowOut` variable, upon which all other constraints rely.

This constraint requires that the output energy of a given process is equal to the product of its input energy and conversion efficiency. Note that the curtailment variable shown below only applies to technologies are members of the curtailment set. As noted in the `Capacity_Constraint`, curtailment is necessary to track explicitly in scenarios that include a high renewable target.

$$\mathbf{FO}_{p,s,d,i,t,v,o} + \mathbf{CUR}_{p,s,d,i,t,v,o} = EFF_{i,t,v,o} \cdot \mathbf{FI}_{p,s,d,i,t,v,o} \quad (5.5)$$

$$\forall \{p, s, d, i, t, v, o\} \in \Theta_{\text{ProcessBalance}}$$

`temoa_rules.CommodityBalance_Constraint` (M, p, s, d, c)

Where the Demand constraint (5.4) 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.

$$\sum_{I,T,V} \mathbf{FO}_{p,s,d,i,t,v,c} = \sum_{T,V,O} \mathbf{FI}_{p,s,d,c,t,v,o} \quad (5.6)$$

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

5.5.3 Physical and Operational Constraints

These three constraints fine-tune the algebraic map created by the three previous constraints, based on various physical and operational real-world phenomena.

`temoa_rules.BaseloadDiurnal_Constraint` (M, 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.

$$SEG_{s,D_0} \cdot \mathbf{ACT}_{p,s,d,t,v} = SEG_{s,d} \cdot \mathbf{ACT}_{p,s,D_0,t,v} \quad (5.7)$$

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

`temoa_rules.DemandActivity_Constraint` ($M, p, s, d, t, v, dem, s_0, d_0$)

For end-use demands, it is unreasonable to let the optimizer only allow use in a single time slice. 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.

$$DEM_{p,s,d,dem} \cdot \sum_I \mathbf{FO}_{p,s_0,d_0,i,t,v,dem} = DEM_{p,s_0,d_0,dem} \cdot \sum_I \mathbf{FO}_{p,s,d,i,t,v,dem} \quad (5.8)$$

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

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

This constraint tracks the amount of storage assuming ordered time slices. The storage unit is initialized at a user-specified charge level (0-100%) in the first time slice of 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 be zeroed out.

`temoa_rules.StorageEnergyUpperBound_Constraint` (M, 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.

`temoa_rules.StorageChargeRate_Constraint` (M, 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.

`temoa_rules.StorageDischargeRate_Constraint` (M, 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.

`temoa_rules.StorageThroughput_Constraint` (M, 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.

`temoa_rules.RampUpDay_Constraint` (M, 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 \mathbf{T}^{ramp} . 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 (5.9). Therefore, the ramp rate constraints between two adjacent seasons are represented in (5.10).

In Equation (5.9) and (5.10), we assume $\mathbf{S} = \{s_i, i = 1, 2, \dots, ns\}$ and $\mathbf{D} = \{d_i, i = 1, 2, \dots, nd\}$.

$$\frac{\text{ACT}_{p,s,d_{i+1},t,v}}{\text{SEG}_{s,d_{i+1}} \cdot C2A_t} - \frac{\text{ACT}_{p,s,d_i,t,v}}{\text{SEG}_{s,d_i} \cdot C2A_t} \leq r_t \cdot \text{CAPAVL}_{p,t} \quad (5.9)$$

$$\forall p \in \mathbf{P}^o, s \in \mathbf{S}, d_i, d_{i+1} \in \mathbf{D}, t \in \mathbf{T}^{ramp}, v \in \mathbf{V}$$

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

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

$$\frac{\text{ACT}_{p,s_{i+1},d_1,t,v}}{\text{SEG}_{s_{i+1},d_1} \cdot C2A_t} - \frac{\text{ACT}_{p,s_i,d_{nd},t,v}}{\text{SEG}_{s_i,d_{nd}} \cdot C2A_t} \leq r_t \cdot \text{CAPAVL}_{p,t} \quad (5.10)$$

$$\forall p \in \mathbf{P}^o, s_i, s_{i+1} \in \mathbf{S}, d_1, d_{nd} \in \mathbf{D}, t \in \mathbf{T}^{ramp}, v \in \mathbf{V}$$

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

Similar to Equation (5.9), we use Equation (5.11) to limit ramp down rates between any two adjacent time slices.

$$\frac{\text{ACT}_{p,s,d_{i+1},t,v}}{\text{SEG}_{s,d_{i+1}} \cdot C2A_t} - \frac{\text{ACT}_{p,s,d_i,t,v}}{\text{SEG}_{s,d_i} \cdot C2A_t} \geq -r_t \cdot \text{CAPAVL}_{p,t} \quad (5.11)$$

$$\forall p \in \mathbf{P}^o, s \in \mathbf{S}, d_i, d_{i+1} \in \mathbf{D}, t \in \mathbf{T}^{ramp}, v \in \mathbf{V}$$

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

Similar to Equation (5.10), we use Equation (5.12) to limit ramp down rates between any two adjacent seasons.

$$\frac{\text{ACT}_{p,s_{i+1},d_1,t,v}}{\text{SEG}_{s_{i+1},d_1} \cdot C2A_t} - \frac{\text{ACT}_{p,s_i,d_{nd},t,v}}{\text{SEG}_{s_i,d_{nd}} \cdot C2A_t} \geq -r_t \cdot \text{CAPAVL}_{p,t} \quad (5.12)$$

$$\forall p \in \mathbf{P}^o, s_i, s_{i+1} \in \mathbf{S}, d_1, d_{nd} \in \mathbf{D}, t \in \mathbf{T}^{ramp}, v \in \mathbf{V}$$

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

During each period p , the sum of the available capacity of all reserve technologies $\sum_{t \in T^{res}} \text{CAPAVL}_{p,t}$, which are defined in the set \mathbf{T}^{res} , should exceed the peak load by RES_z , the regional 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.

$$\sum_{t \in T^{res}} CC_t \cdot \text{CAPAVL}_{p,t} \cdot \text{SEG}_{s^*,d^*} \cdot C2A_t \geq \sum_{t \in T^{res}} \sum_{t \in v^{vintage}} \text{ACT}_{p,s,d,t,v} \cdot (1 + RES_z) \quad (5.13)$$

$$\forall p \in \mathbf{P}^o, z \in \mathbf{C}^{zone}$$

5.5.4 Objective Function

temoa_rules.**TotalCost_rule** (M)

Using the Activity 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.

$$C_{loans} = \sum_{t,v \in \Theta_{IC}} \left(\left[IC_{t,v} \cdot LA_{t,v} \cdot \frac{(1 + GDR)^{P_0 - v + 1} \cdot (1 - (1 + GDR)^{-LLN_{t,v}})}{GDR} \cdot \frac{1 - (1 + GDR)^{-LPA_{t,v}}}{1 - (1 + GDR)^{-LP_{t,v}}} \right] \cdot \text{CAP}_{t,v} \right) \quad (5.14)$$

Note that capital costs ($IC_{t,v}$) 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 $\frac{GDR}{1-(1+GDR)^{-LP_{t,v}}} \cdot \frac{1-(1+GDR)^{-LPA_{t,v}}}{GDR}$. The product simplifies to $\frac{1-(1+GDR)^{-LPA_{t,v}}}{1-(1+GDR)^{-LP_{t,v}}}$, where $LPA_{t,v}$ represents the active lifetime of a process (t, v) before the end of the model horizon, and $LP_{t,v}$ represents the full lifetime of a process (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.

$$C_{fixed} = \sum_{p,t,v \in \Theta_{FC}} \left(\left[FC_{p,t,v} \cdot \frac{(1+GDR)^{P_0-p+1} \cdot (1-(1+GDR)^{-MPL_{t,v}})}{GDR} \right] \cdot \mathbf{CAP}_{t,v} \right) \quad (5.15)$$

$$C_{variable} = \sum_{p,t,v \in \Theta_{VC}} \left(MC_{p,t,v} \cdot \frac{(1+GDR)^{P_0-p+1} \cdot (1-(1+GDR)^{-MPL_{t,v}})}{GDR} \cdot \mathbf{ACT}_{t,v} \right) \quad (5.16)$$

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, 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.

$$\begin{aligned} \mathbf{CAP}_{t,v} &= ECAP_{t,v} \\ \forall \{t, v\} &\in \Theta_{ExistingCapacity} \end{aligned} \quad (5.17)$$

`temoa_rules.EmissionLimit_Constraint` (M, 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.

$$\sum_{I,T,V,O|e,i,t,v,o \in EAC_{ind}} (EAC_{e,i,t,v,o} \cdot \mathbf{FO}_{p,s,d,i,t,v,o}) \leq ELM_{p,e} \quad (5.18)$$

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

`temoa_rules.GrowthRateConstraint_rule` (M, p, t)

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

$$CAPAVL_{p_i,t} \leq GRM \cdot CAPAVL_{p_{i-1},t} + GRS, \quad (5.19)$$

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

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, p, t)

The MaxActivity sets an upper bound on the activity from a specific technology. Note that the indices for these constraints are period and tech, not tech and vintage.

$$\sum_{S,D,V} \mathbf{ACT}_{p,s,d,t,v} \leq \mathbf{MAXACT}_{p,t} \quad (5.20)$$

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

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

The MinActivity sets a lower bound on the activity from a specific technology. Note that the indices for these constraints are period and tech, not tech and vintage.

$$\sum_{S,D,V} \mathbf{ACT}_{p,s,d,t,v} \geq \mathbf{MINACT}_{p,t} \quad (5.21)$$

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

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, which determines what technology activity share can count towards the constraint.

$$\sum_{S,D,T,V} \mathbf{ACT}_{p,s,d,t,v} \geq \mathbf{MGGT}_{p,g} \quad (5.22)$$

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

where g represents the assigned technology group and $MGGT$ refers to the MinGenGroupTarget parameter.

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

See MaxCapacity_Constraint

temoa_rules.**MaxCapacity_Constraint** (M, 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 period and tech, not tech and vintage.

$$\mathbf{CAPAVL}_{p,t} \leq \mathbf{MAX}_{p,t} \quad (5.23)$$

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

temoa_rules.**MinCapacity_Constraint** (M, 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 period and tech, not tech and vintage.

$$\mathbf{CAPAVL}_{p,t} \geq \mathbf{MIN}_{p,t} \quad (5.24)$$

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

temoa_rules.**ResourceExtraction_Constraint** (M, 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.

$$\sum_{S,D,I,t \in Tr, V} \mathbf{FO}_{p,s,d,i,t,v,c} \leq \mathbf{RSC}_{p,c} \quad (5.25)$$

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

temoa_rules.**TechInputSplit_Constraint** (M, 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.

`temoa_rules.TechOutputSplit_Constraint` (M, 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:

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

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. The constraint is formulated as follows:

$$\sum_I \mathbf{FO}_{p,s,d,i,t,v,o} \geq \mathbf{SPL}_{p,t,o} \cdot \mathbf{ACT}_{p,s,d,t,v} \quad (5.26)$$

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

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 $\frac{1}{\text{elceff}}$ 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 avenue to describe mathematical optimization models for a computational context, and allow for abstract model⁷ 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 people. 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 a recently developed 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 constraint (5.4):

$$\sum_{I,T,V} \mathbf{FO}_{p,s,d,i,t,v,dem} \geq DEM_{p,dem} \cdot DSD_{s,d,dem}$$

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

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.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, p, s, d, dem ):
2     if (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     # store the summation into the local variable 'supply' for later reference
6     supply = sum(
7         M.V_FlowOut[p, s, d, S_i, S_t, S_v, dem]
8
9         for S_t, S_v in M.commodityUStreamProcess[p, dem]
10        for S_i in M.ProcessInputsByOutput[p, S_t, S_v, dem]
11    )
12
13    # The '=' operator creates (in this case) a "Equal" *object*, not a
14    # True/False value as a Python programmer might expect; the intermediate
15    # variable 'expr' is thus not strictly necessary, but we leave it as reminder
16    # of this potentially confusing behavior
17    expr = (supply = M.Demand[p, s, d, dem])
18
19    # finally, return the new "Equal" object (not boolean) to Pyomo
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 `time_optimize`, the second from `time_season`, the

third from `time_of_day`, and the fourth from `commodity_demand`. Though one could choose `a`, `b`, `c`, and `d` (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 four index sets, so Pyomo will naively call this function for every possible combination of tuple $\{p, s, d, dem\}$. 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 6 through 12 are a single *source-line* that we split over 7 lines for clarity. These lines implement the summation of the Demand constraint, summing over all technologies, vintages, and the inputs that generate the end-use demand `dem`. Note that the sum is performed with sparse indices, which are returned from dictionaries created in `temoa_initialize.py`.

Lines 6 through 12 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 )
```

While both implementations have the same number of lines, this last one 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` variable, line 18 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 19.

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

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.

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_psdc = Set( dimen=4, rule=DemandConstraintIndices )
2 # ...
3 M.DemandConstraint = Constraint( M.DemandConstraint_psdc, rule=Demand_Constraint )

```

As discussed above, the DemandConstraint is only valid for certain $\{p, s, d, dem\}$ tuples. Since the modeler can specify demand distribution per commodity (necessary to model demands like heating, that do not make sense in the summer), Temoa must ascertain the exact valid tuples. We have implemented this logic in the function `DemandConstraintIndices` in `temoa_initialize.py`. Thus, Line 1 tells Pyomo to instantiate `DemandConstraint_psdc` as a Set of 4-length tuples indices (`dimen=4`), 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_psdc` created, we can now use it in place of the four 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 `_psdc` 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_psdc` implies that this set has a dimensionality of 4, and (following the *namingscheme*) the first index of each tuple will be an element of `time_optimize`, the second an element of `time_season`, the third an element of `time_of_day`, and the fourth a commodity. From the contextual information that this is the Demand constraint, one can assume that the `c` represents an element from `commodity_demand`.

Over a sparse-index, the constraint implementation changes only slightly:

in `temoa_rules.py` (actual implementation)

```

1 def Demand_Constraint ( M, p, s, d, dem ):
2     supply = sum(
3         M.V_FlowOut[p, s, d, S_i, S_t, S_v, dem]
4         for S_t, S_v in M.commodityUStreamProcess[p, dem]
5         for S_i in M.ProcessInputsByOutput[p, S_t, S_v, dem]
6     )
7
8     DemandConstraintErrorCheck ( supply, dem, p, s, d )
9
10    expr = (supply = M.Demand[p, dem] * M.DemandSpecificDistribution[s, d, dem])
11    return expr

```

As this constraint is guaranteed to be called only for necessary demand constraint indices, there is no need to check for the existence of a tuple in the Demand parameter. The only other change is the error check on line 10. This function is defined in `temoa_initialize.py`, and simply ensures that at least one process supplies the demand `dem` in time slice $\{p, s, d\}$. If no process supplies the demand, then it quits computation immediately (as opposed to completing a potentially lengthy model generation and waiting for the solver to recognize the infeasibility of the model). Further, the function lists potential places for the modeler to look to correct the problem. This last capability is subtle, but in practice extremely useful while building and debugging a model.

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 (5.4). 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 alternatives (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
```

(continues on next page)

(continued from previous page)

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:

```
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
```

(continues on next page)

(continued from previous page)

```
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 than 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
```

(continues on next page)

(continued from previous page)

```

>>> 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.
```

(continues on next page)

(continued from previous page)

```
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 contributors may unilaterally change the license of Temoa, unless **all** contributors 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

- A**
Activity_Constraint () (in module *temoa_rules*), 29
- B**
BaseloadDiurnal_Constraint () (in module *temoa_rules*), 30
- C**
Capacity_Constraint () (in module *temoa_rules*), 29
CapacityAvailableByPeriodAndTech_Constraint () (in module *temoa_rules*), 29
CommodityBalance_Constraint () (in module *temoa_rules*), 30
- D**
Demand_Constraint () (in module *temoa_rules*), 30
DemandActivity_Constraint () (in module *temoa_rules*), 31
- E**
EmissionLimit_Constraint () (in module *temoa_rules*), 33
ExistingCapacity_Constraint () (in module *temoa_rules*), 33
- G**
GrowthRateConstraint_rule () (in module *temoa_rules*), 33
- M**
MaxActivity_Constraint () (in module *temoa_rules*), 33
MaxCapacity_Constraint () (in module *temoa_rules*), 34
MaxCapacitySet_Constraint () (in module *temoa_rules*), 34
MinActivity_Constraint () (in module *temoa_rules*), 34
MinActivityGroup_Constraint () (in module *temoa_rules*), 34
MinCapacity_Constraint () (in module *temoa_rules*), 34
- P**
ProcessBalance_Constraint () (in module *temoa_rules*), 30
- R**
RampDownDay_Constraint () (in module *temoa_rules*), 32
RampDownSeason_Constraint () (in module *temoa_rules*), 32
RampUpDay_Constraint () (in module *temoa_rules*), 31
RampUpSeason_Constraint () (in module *temoa_rules*), 32
ReserveMargin_Constraint () (in module *temoa_rules*), 32
ResourceExtraction_Constraint () (in module *temoa_rules*), 34
- S**
StorageChargeRate_Constraint () (in module *temoa_rules*), 31
StorageDischargeRate_Constraint () (in module *temoa_rules*), 31
StorageEnergy_Constraint () (in module *temoa_rules*), 31
StorageEnergyUpperBound_Constraint () (in module *temoa_rules*), 31
StorageThroughput_Constraint () (in module *temoa_rules*), 31
- T**
TechInputSplit_Constraint () (in module *temoa_rules*), 34
TechOutputSplit_Constraint () (in module *temoa_rules*), 35
TotalCost_rule () (in module *temoa_rules*), 32