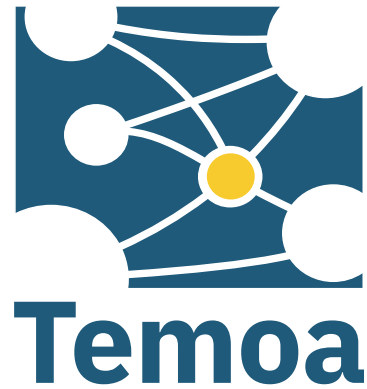




Tools for Energy Model Optimization and Analysis (Temoa)

Release 4.0.0a1.dev20251201

TemoaProject Team

Apr 02, 2026

GETTING STARTED

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	2
2	Quick Start	3
2.1	Installing Software Elements	3
2.2	Solvers	4
2.3	Running Temoa	4
3	Database Construction	7
4	Data Quality	11
4.1	Price Checking	11
4.2	Source Tracing	11
5	Commodity Network	13
5.1	Network Flow	13
5.2	Source Tracing	14
5.3	Tech Suppression	15
6	Visualization	17
6.1	Network Diagrams	17
7	Mathematical Formulation	19
7.1	Temoa Notation	20
7.2	Treatment of Time	21
7.3	Sets	23
7.4	Parameters	27
7.5	Decision Variables	42
7.6	Equations	45
7.7	General Caveats	67
8	Computational Implementation	69
8.1	Anatomy of a Constraint	69
8.2	A Word on Verbosity	73
8.3	Execution Flow	74
8.4	Project Structure	75
8.5	The Bleeding Edge	76
8.6	Ruff Formatting	78

8.7	Indentation: Tabs and Spaces	79
8.8	End of Line Whitespace	79
8.9	Maximum Line Length	79
8.10	Blank Lines	79
8.11	Encodings	80
8.12	Naming Conventions	80
8.13	In-line Implementation Conventions	80
8.14	Miscellaneous Style Conventions	81
8.15	Patches and Commits to the Repository	82
9	Modeling to Generate Alternatives (MGA)	85
9.1	Hull Expansion	85
9.2	Single-Vector MGA (SVMGA)	86
9.3	Parallel Execution and Solver Options	86
9.4	Outputs	87
10	Monte Carlo Simulation	89
10.1	Overall Framework	89
10.2	Configuration	89
10.3	Input File Format	89
10.4	Parallel Execution	90
10.5	Outputs	91
11	Myopic Optimization	93
11.1	Overall Framework	93
11.2	Configuration	93
11.3	Evolving vs. Non-Evolving Mode	94
11.4	How it Works	95
11.5	Myopic Efficiency Table	95
11.6	Backtracking and Roll-back	96
11.7	Notes and Caveats	96
12	Stochastic Programming	97
12.1	Dependencies	97
12.2	Configuration	97
12.3	How it Works	98
12.4	Limitations	99
13	Units Checking	101
13.1	Unit Propagation to Outputs	101
13.2	Overview	101
13.3	Enabling Unit Checking	102
13.4	How It Works	102
13.5	Expressing Units	102
13.6	Common Footguns and Pitfalls	103
13.7	Testing and Troubleshooting	105
13.8	Tables Checked	106
13.9	Best Practices	107
13.10	Quick Reference	107
13.11	See Also	108
14	References	109
	Bibliography	111

PREFACE

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

Besides this documentation, there are a couple other sources for Temoa-oriented information. For questions, bug reports, or feature requests, please use our [GitHub Issues](#). 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 [GitHub Issues](#)) 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) developed over many years to support transparent, data-driven analysis of energy systems. ESOMs serve as an important planning tool because they allow users to examine energy futures using a self-consistent framework for evaluation. Temoa is implemented as a linear program that minimizes the total cost of energy supply by optimizing the installation and operation of energy technologies over a user-defined planning horizon. Energy supply must meet end-use demands subject to physical and technical constraints governing system operation, along with user-defined policy scenarios.

The energy system on which Temoa and other ESOMs operate can be visualized as a directed graph. Primary energy sources represent the points of origin, which are transformed by a network of energy conversion and delivery technologies, and ultimately produce consumable energy commodities that satisfy end-use demands. [\[esom_definition\]](#). Temoa provides tools to explicitly represent this network, visualize system structure, and trace energy flows through time.

A defining strength of Temoa is its flexible treatment of time. Users may define arbitrary model periods of varying length and represent intra-period operations using seasonal and time-of-day slices, full chronological hours, or representative days. Capacity expansion can be solved under perfect foresight or using a rolling-horizon. In addition, Temoa supports technology vintaging, separate loan periods and physical lifetimes, and both global and technology-specific discount rates. Beyond deterministic optimization, Temoa supports stochastic optimization as well as modeling-to-generate alternatives (MGA) to explore near-optimal solution spaces. All of Temoa's features were driven by specific analytic needs over a decade of model development and policy-focused application.

Temoa is implemented within an open-source software stack and is released under the MIT license, with source code available on GitHub [\[open_source_realities\]](#). The model is written in Python and seamlessly integrates with the broader Python ecosystem. Input data are stored in a relational SQLite database, enabling transparency, reproducibility, and easy modification. The model maintains a strict distinction between source code and the input data on which it operates. The model can be executed on single machines, multi-core systems, or high-performance computing environments.

The name Temoa (Tools for Energy Model Optimization and Analysis) reflects the project's broader scope. The platform comprises four interrelated components: the underlying mathematical formulation, its software implementation, a suite of supporting tools for data management, analysis, and visualization, and an online presence that supports documentation, dissemination, and community engagement. Together, these elements are designed to foster collaboration, extensibility, and trust in energy system modeling results.

1.2 Why Temoa?

In 2009, when the idea for Temoa was born, most options for energy systems modeling were geared towards government institutions that could afford the expensive commercial software licenses. Closed source code and data also meant that it was impossible for third parties to verify published model results, even though those results were being used to inform public policy decisions involving significant transfers of wealth and direct consequences for people's lives. In addition, models were typically used to run a limited number of scenarios that did not address the true underlying uncertainty about the future.

Today's vibrant open source energy modeling community did not exist at that time. We were motivated to build Temoa around three high-level objectives: (1) make the model code and data open source to enable third party replication of results, (2) use an open source software stack to minimize the barriers to entry in energy modeling, and (3) build a toolkit to evaluate future uncertainty in different ways, depending on the question at hand.

Temoa remains one of the most fully-featured, open source energy system models focused on projecting changes across the whole energy system.

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."

TEMŌĀ *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](#).

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.

Using pip:

The standard way to install Temoa:

First, it is highly recommended to use a Python virtual environment to manage dependencies:

```
# Linux / macOS
$ python -m venv temoa_env
$ source temoa_env/bin/activate

# Windows (Command Prompt)
> python -m venv temoa_env
> temoa_env\Scripts\activate

# Windows (PowerShell)
PS> python -m venv temoa_env
PS> .\temoa_env\Scripts\Activate.ps1
```

Then, install Temoa:

```
# Install from PyPI
$ pip install temoa

# Get started
$ python -m temoa tutorial my_first_model
$ python -m temoa run my_first_model.toml
```

Using uv (Alternative):

For faster dependency resolution:

```
# Install uv (Linux / macOS)
$ curl -LsSf https://astral.sh/uv/install.sh | sh

# Install uv (Windows)
PS> powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

(continues on next page)

(continued from previous page)

```
# Create a uv initialized virtual environment
$ uv init <dir_name>
$ cd <dir_name>

# add Temoa from PyPI to pyproject.toml
$ uv add temoa

# Get started
$ uv run temoa tutorial my_first_model
$ uv run temoa run my_first_model.toml
```

For Contributors (Development Installation):

If you want to contribute to Temoa or modify the code:

```
# Clone the repository
$ git clone https://github.com/TemoaProject/temoa.git
$ cd temoa

# Install in development mode with uv (recommended)
$ uv sync --all-extras --dev

# Install pre-commit hooks
$ uv run pre-commit install
```

For detailed contribution guidelines, see CONTRIBUTING.md in the repository.

2.2 Solvers

A few notes 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.

For smaller models, Temoa has been tested with the [HiGHS](#) solver. HiGHS is automatically available when you install Temoa and requires no additional setup.

2.3 Running Temoa

To run the model, a configuration (config) file is needed. An example config file called `config_sample.toml` resides within the `data_files/my_configs` 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, (3) specify the solver to use, (4) return the log file produced during model execution, (5) return the lp file utilized by the solver, and (6) to execute several of the modeling extensions.

```
$ temoa run config.toml
```

For general help, use `--help`:

```
$ temoa --help
usage: temoa [-h] [--version] [--how-to-cite] COMMAND ...

Tools for Energy Model Optimization and Analysis (Temoa)
```

options:

-h, --help	show this help message and exit
--version	Show version information
--how-to-cite	Show citation information

commands:

COMMAND

run	Run a Temoa model
validate	Validate a configuration file
check-units	Check unit consistency in a database
migrate	Migrate a database to the latest schema
tutorial	Create tutorial files

DATABASE CONSTRUCTION

Input datasets in Temoa are stored in a relational database management system. For those unfamiliar with databases, you can think of them as collections of tables. Within each table, a ‘primary key’ uniquely identifies each row. A ‘foreign key’ is a column in one table that references the primary key of another table, thereby establishing relationships between tables and ensuring data consistency across the database.

The following Entity-Relationship (ER) diagram provides a visual overview of the Temoa v4 database schema and the relationships between its various tables:

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 `time_period` table used to define the `test_system` dataset:

```
1 CREATE TABLE time_period
2 (
3     sequence INTEGER UNIQUE,
4     period   INTEGER
5     PRIMARY KEY,
6     flag     TEXT
7     REFERENCES time_period_type (label)
8 );
9 INSERT INTO "time_period" VALUES(1,2015,'e');
10 INSERT INTO "time_period" VALUES(2,2020,'f');
11 INSERT INTO "time_period" VALUES(3,2025,'f');
12 INSERT INTO "time_period" VALUES(4,2030,'f');
13 INSERT INTO "time_period" VALUES(5,2035,'f');
```

The first line creates the table. **Lines 3-7** define the columns within this table. Note that `period` is the primary key. Therefore, the same time period cannot be entered twice; each name must be unique. **Line 7**, which helps define the `flag` column, declares a foreign key reference to the `label` column of the `time_period_type` table. As a result, if the user tries to enter a label in this table that does not exist in the `time_period_type` table, it will fail with an error. This foreign key reference ensures that the modeler doesn’t accidentally type the wrong label in this table. For context, there are two basic types of time periods in Temoa, `e`, which defines pre-existing periods, and `f`, which defines future time periods that are to be optimized.

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_type`: Need to identify which type of commodity. Do NOT change these abbreviations.
- `technology_type`: Need to identify which type of technology. Do NOT change these abbreviations. Categorizing and sub-categorizing can be done in the Technology table itself.
- `time_period_type`: 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

- `commodity`: list of commodities used within the database
- `technology`: list of technologies used within the database
- `time_period`: list of both past and future time periods considered in the database
- `time_season`: seasons modeled in the database (also contains segment fractions)
- `time_of_day`: time of day segments modeled in the database

Group 3: parameters used to define processes within Temoa

- `metadata_real` (`global_discount_rate`)
- `demand`
- `demand_specific_distribution`
- `efficiency`
- `existing_capacity`
- `capacity_factor_tech`
- `capacity_factor_process` (only if CF varies by vintage; overwrites `capacity_factor_tech`)
- `capacity_to_activity`
- `cost_fixed`
- `cost_invest`
- `cost_variable`
- `emission_activity`
- `lifetime_tech`
- `lifetime_process` (only if LT varies by vintage; overwrites `lifetime_tech`)

Group 4: parameters used to define constraints within Temoa (non-exhaustive)

- `limit_activity`
- `limit_capacity`
- `limit_emission`
- `limit_growth_capacity`
- `limit_new_capacity`
- `limit_resource`
- `limit_tech_input_split`
- `limit_tech_output_split`

For help getting started, consider using the `temoa tutorial <model_name>` command to generate a template project or inspect the example SQL file at `temoa/tutorial_assets/utopia.sql`. To begin building your own database file, use `temoa/db_schema/temoa_schema_v4.sql`, which is a database file with the requisite structure but no data added. We recommend leaving the database structure intact, and simply adding data to the schema file, or constructing an empty database from the schema file and then using a script or database editor to import data. 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
```

Note

The command above using the `<` operator for input redirection is supported in Linux, macOS, and the Windows Command Prompt. Note that PowerShell does not support the Unix-style `<` redirection operator. In PowerShell, you can achieve the same result by piping the file content into `sqlite3`:

```
Get-Content my_database.sql | sqlite3 my_database.sqlite
```

Alternatively, you can load a SQL file from within the `sqlite3` interactive interface using the `.read` command:

```
sqlite3 my_database.sqlite
sqlite> .read my_database.sql
```

When running on Windows, be sure to use the correct path separators (`\`) if you provide absolute paths to your files.

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

DATA QUALITY

In addition to numerous internal checks, Temoa (optionally) employs two quality checks on data read in from the database. The outputs (actions and warnings) generated by these processes are reported in the log file for the run.

Both of the checks below can be run to QA data by running the model in *CHECK* mode and inspecting the log file. During *CHECK* mode runs, no solve is attempted on the model.

4.1 Price Checking

The “price checker” reviews cost data in the 3 cost tables and considers technology lifetime. It screens for possible inconsistencies that would corrupt output quality. Larger models may have well over 100K cost entries and an overlooked investment cost for a particular vintage tech in a particular region could easily be overlooked. Price checks performed/reported:

1. **Missing Costs (Check 0):** This check looks for technologies that have no fixed/invest/variable costs at all. Other checks are more discriminating, so this check is only reported when Temoa is run in *debug* mode by using the *-d* flag on the run command.
2. **Missing Fixed/Investment Costs (Check 1a):** This check identifies technologies that are *not* flagged as *uncapacitated* with neither a fixed or investment cost associated. These *might* be problematic for solve because the model minimizes cost, so capacity in these technologies would be free. *uncapacitated* technologies have no capacity measure, so fixed/investment costs are prohibited for them and that is checked elsewhere.
3. **Inconsistent Fixed/Investment Cost (Check 1b):** This check looks for inconsistent application of fixed or base costs in the “base” or vintage year across all vintages and regions. So, if a tech has a fixed cost in some particular region and vintage year, but not in all, it will be flagged as a likely omission.
4. **Inconsistent Fixed & Variable Costs (Check 2):** This check identifies techs that have inconsistencies in the application of fixed - variable costs. Techs that have *any* fixed cost for a particular [region, tech, vintage] process, but do not have entries that match the variable cost entries for the same process are flagged, and vice-versa. This would hopefully identify an accidental omission of some of the fixed/var costs for processes that have at least 1 entry for either.
5. **Lifetime Costing (Check 3):** This check identifies costs that fall short or are missing during the process’s lifetime. If a process has a variable cost in *any* year during the lifetime, but not all years, it is flagged. Same for fixed cost.
6. **Uncapacitated Tech Costs:** Any technology flagged as *uncapacitated* will trigger warnings here if it has any fixed/invest costs.

4.2 Source Tracing

Temoa works backwards from demands to identify chains of technologies required to meet the demand. Source Tracing is designed to ensure that this backward tracing from demands describes a proper commodity network without gaps

that might allow intermediate commodities to be treated as a free “source” commodity. Further description of possible network problems is included in the [Commodity Network](#) section.

Source Tracing pre-builds the entire commodity network in each region-period contained in the data and analyzes it for “orphans” which likely represent gaps in the network that would lead to erroneous output data. The operation is enabled by tagging foundational commodities for which there are no predecessors as “source” commodities in the *Commodity* database table with an *s* tag. Orphans (or chains of orphans) on either the demand or supply side are reported and *suppressed* in the data to prevent network corruption. Additionally, Temoa performs cycle detection on the commodity network to identify circular dependencies that could lead to non-convergence or erroneous results. Users can configure the cycle detection behavior using the following settings:

- **cycle_count_limit:** Limits the number of cycles reported in the log. A value of *-1* allows unbounded detection, *0* causes the system to log an error on the first detected cycle and then suppresses further cycle reports for the remainder of the run (without terminating execution), and a positive integer sets a specific limit. Default is 100.
- **cycle_length_limit:** Minimum length of cycles to report. This can be used to filter out small, expected circularities if necessary. Default is 1. The length limit is inclusive, so a cycle of length 1 is a self-loop, and a cycle of length *n* has *n* unique nodes.

Note that the myopic mode *requires* the use of Source Tracing to ensure accuracy as some orphans may be produced by endogenous decisions in myopic runs.

COMMODITY NETWORK

This documentation segment highlights some of the techniques to maintain the integrity of the commodity flow network within Temoa. The modeler is advised to be aware of how Temoa processes flow of energy within an energy system and how escapes in network integrity can lead to erroneous results.

5.1 Network Flow

In a standard, static, and lossless $s-t$ network, flow balance constraints fall into place naturally. The input flow at the source, s is set to equal the output flow at termination point t and for all intermediary nodes, flow in is constrained to equal flow out. Temoa's energy network, however, is not static and has losses in the form of efficiency losses within the `tech` processes that serve as the links between `commodity` nodes. When a model is solved over several time periods with "perfect foresight" the network may look quite different in each region and time period based on endogenous selections of new technologies, lifetime expirations, etc. This diversification of networks is compounded when running the model myopically where previous non-selections remove options from the decision space in future periods, outside of the modeler's control.

When developing data for a model to run with perfect foresight, or any of the more advanced modes, it is incumbent upon the modeler to provide the vintages, technologies, lifespans, etc. in order to allow Temoa to build a proper network. Some explanation of how Temoa balances flow is in order to help avoid pitfalls....

The energy network in each region and period is built dynamically from the `techs` that are identified as existing capacity and whatever the model has ability to build from elements in the optimization window identified in the `efficiency` table. In order to enforce conservation of flow constraints, the model identifies output flows (commodities), including the demand commodities and inventories all possible `techs` (existing or available) that *could* produce that commodity from any input commodity and enforces flow balance by requiring capacity in one of the available technologies, so that *some tech* must provide the flow to support that output. This is the basis for satisfying demands at the termination of the network.

If at any point along the chain there are no `techs` available to produce the commodity in question, the model *assumes this is a base or source commodity* for flow balance purposes, which can lead to some intermediate physical commodities erroneously being provided to the model without a preceding processing chain if none is available. This can inadvertently come about from a variety of causes including:

- Failure to provide a linking technology in any region-period
- A technology that expires in a particular period with no replacement/alternate pathway provided
- Non-selection in a prior myopic period

These failures can be difficult to diagnose in large models with many periods/regions/technologies and serve as the motivation to provide source-tracing (described in the next section) to help enforce network integrity.

5.2 Source Tracing

Source tracing from Demand back to a labeled “source” capacity is now available in Version 3.0+ of Temoa. During pre-processing, before the model is built, Temoa can identify breaks in the network and can filter out problematic data before the model is loaded/built. Temoa can also check network integrity on a built model before solve is initiated. Currently, discrepancies are noted in the log file for the model. The user can also request network plots, which are browser-capable html files that can be used in diagnosis of problem areas. The general intent is to ensure that all flows to Demand commodities can be cleanly traced back to original sources.

An example:

Consider the simple network below with one source, one demand, and intermediate physical commodities {P1, P2, P3, P4} and the connective technologies T1 through T6. This well-connected network works as intended and the singular demand is traceable via either path back to source.

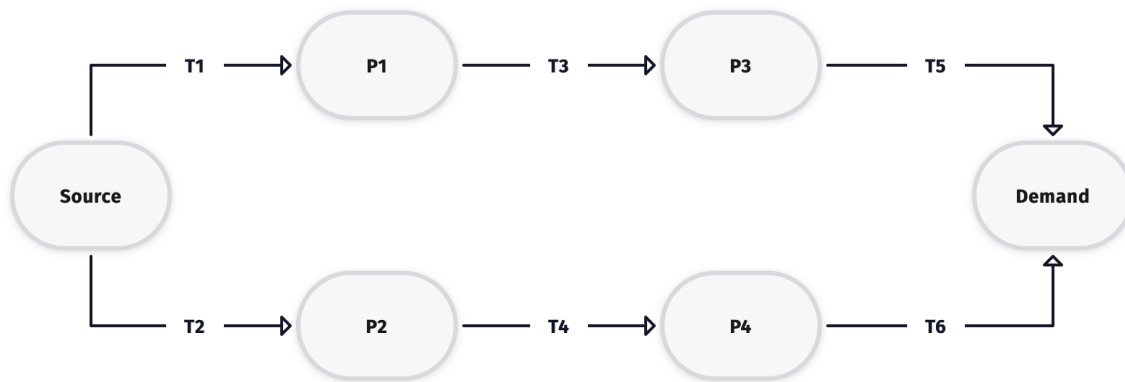


Fig. 1: Good Network

A defective network (shown below) may occur for a several reasons, as cited in the previous section. Suppose that for some reason T3 is no longer available in this or a subsequent period (never made available, lifetime expiration earlier than other links, not selected by myopic process—which would normally remove the other links as well, unless they had replacement vintages and T3 did not, etc.) Several problems now exist:

1. Supply side orphans. Technology T1 is now a “supply side” orphan, which shouldn’t cause model problems, but represents bloat in the model. Legacy (pre version 3.0) Temoa does screen for unused outputs (like P1 in this case) that are not used by other processes and are not end demands, but it is currently only done ‘globally’ in all periods/regions. Resultantly, this orphan may not trigger a model error if it were used in another region/period. These will now generate **WARNING** level log entries with source tracing. It *could* be a source of unbounded behavior in the case where the modeler attempts to use negative values for costs.
2. Technology T5 and perhaps a now-available new vintage T5' are now “demand-side” orphans. These are problematic and will generate **WARNING** level log entries by source tracing because they would allow a false/unlimited supply of P3 as their inputs.
3. New technology T7 (and any other linkages that are not reachable from either source or demand) are complete orphans. They will generate a **WARNING** level log entry during source tracing.

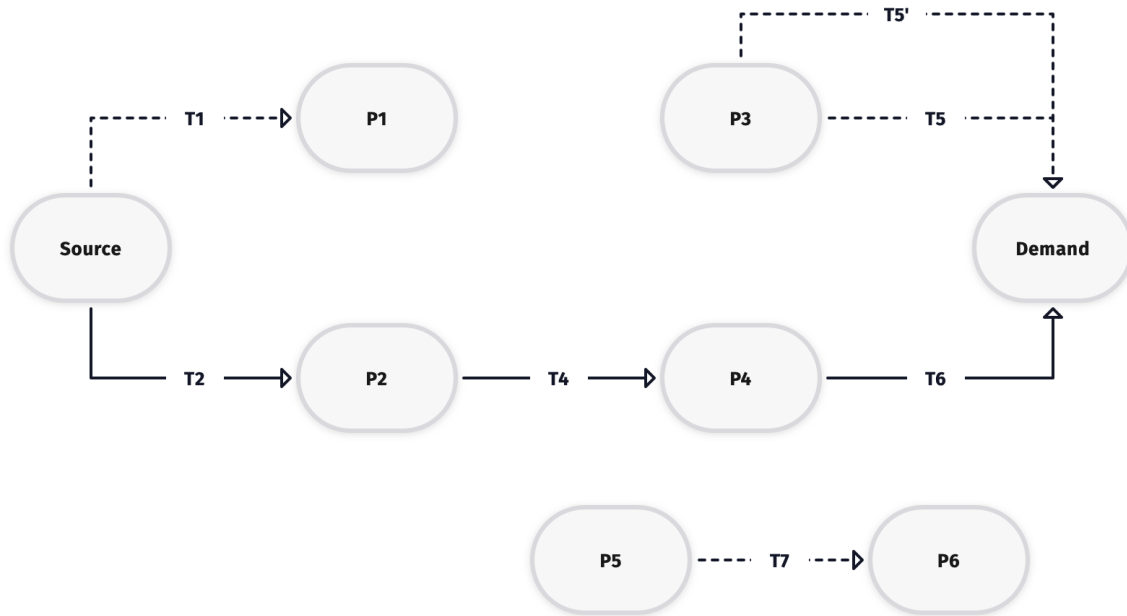


Fig. 2: Bad Network

5.3 Tech Suppression

When source tracing is used, Temoa will attempt to remedy network escapes as described above by suppressing problematic `techs` or chains of technologies. During runs where source tracing is enabled, Temoa's internal `Network Data Manager` will preload essential data in order to analyze all networks (each region/period) within the current optimization window. Network tracing from demands down and sources up reveals `techs` that are "orphaned". These `techs` and any unused commodities, etc. are removed with log entries and the remaining "good" data is used to filter the ingestion of the full model data to support the build.

5.3.1 Basic rules for `tech` Suppression

- All demand side orphans (and chains of orphans) are suppressed
- All supply-side orphans (and chains of orphans) are suppressed
- For technologies that have multiple inputs or outputs, EACH commodity is treated separately (as it is represented in the `efficiency` table.) The modeler is advised to screen the log file for these entries if a particular commodity is essential (a catalyst for example?) to a particular operation.
- For linked technologies, an "all or nothing" logic is implemented such that both the driver and driven technologies must prove independently viable (without regard for the emission linkage) or the pair will be suppressed.
- Currently, exchange technologies are not checked during source tracing. Any `tech` with a region link (denoted with a dash `R1-R2`) provided for the region name is assumed good. It behooves the modeler to ensure that both ends of each link are well-connected in the model and to review their performance in output.

5.3.2 Views across modeling periods

The approach described above is applied individually to each region-period pair within the optimization window. Recall, that for perfect foresight runs, this will be all future periods. For myopic or other limited-visibility runs, this will be applied to each period within view. The orphans are identified on a by period-region basis, but *the suppression is applied for all periods within the view*. For example, if technology `power_plant` has a vintage of 2020 and a lifetime of 40 years covering model periods {2020, 2030, 2040, 2050} and it is not viable in 2030, it will be suppressed and not available in any of the periods.

Currently, for myopic applications, this decision is made on a per-iteration basis, so the technology may be suppressed within the current iteration only. For example, if the myopic view was 1 period, and if the `power_plant` was selected for build in its vintage year, 2020, the modeler may see that it is used in 2020, “suppressed” in 2030 because it may run freely or cause problems in that period, but available again in 2040-2050. This may/may not be desired behavior and the modeler should be aware of these behaviors logged in the log file.

This screening prevents model errors and helps to reduce the size of the model in most cases. These processes with non-utilized outputs may be the result of erroneous data (perhaps a technology to produce liquid hydrogen before there is a user of same) or via myopic actions were a technology in a chain is not selected for build, rendering other processes in the chain irrelevant in a later period. The main goal of using tracing and suppression is to prevent “free” midstream commodities from erroneously feeding processes and prevent free-running techs that might have a negative cost from making the model unbounded or infeasible.

VISUALIZATION

6.1 Network Diagrams

Since the Temoa model consists of an energy network in which technologies are connected by the flow of energy commodities, a directed network graph represents an excellent way to visualize a given energy system representation in a Temoa-compatible input database.

Temoa provides two types of network visualizations:

1. **Interactive HTML Network Graphs** - Dynamic, explorable visualizations showing commodity flows and technology connections
2. **Graphviz Diagrams** - Static SVG/DOT format diagrams showing the energy system structure

6.1.1 Generating Network Visualizations

The easiest way to generate these diagrams is to enable visualization options in your configuration TOML file. Add the following to your config file:

```
# Enable interactive HTML network graphs (requires source_trace = true)
source_trace = true
plot_commodity_network = true

# Enable Graphviz static diagrams
graphviz_output = true
```

When these options are enabled, Temoa will automatically generate visualization files in the output directory during model execution.

Interactive Network Graphs will be created as HTML files (one per time period) that you can open in a web browser. These provide an interactive view where you can:

- Pan and zoom the network
- Click on nodes to see details
- Toggle between commodity-centric and technology-centric views
- Filter by sector using color-coded legends

Graphviz Diagrams will be created as both `.dot` (source) and `.svg` (rendered) files in a subdirectory within your output folder. These provide static visualizations showing:

- Full energy system maps
- Capacity and activity results per model time period
- Technology interconnections via commodity flows

6.1.2 Example Visualizations

Interactive Network Graph

The interactive HTML network graphs provide dynamic exploration with pan, zoom, and filtering capabilities:

Interactive network graph for the ‘utopia’ test system in 1990. You can pan, zoom, click nodes for details, and toggle between commodity-centric and technology-centric views. These files are automatically generated when `source_trace = true` and `plot_commodity_network = true` are set in the configuration file.

Static Graphviz Diagram

Graphviz also generates static SVG diagrams showing the energy system structure:

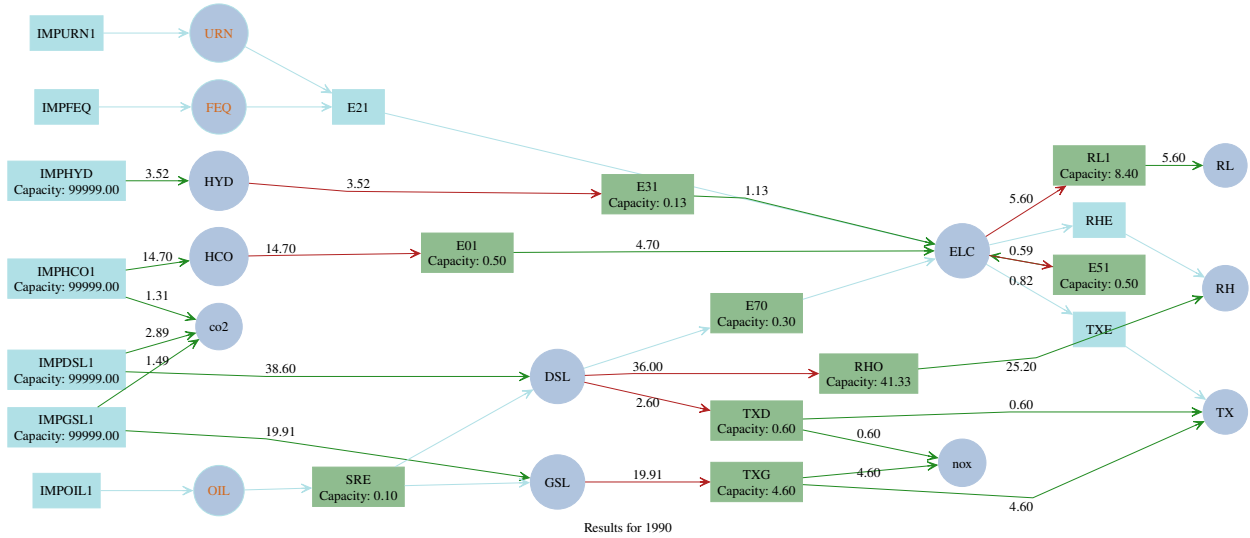


Fig. 1: Static Graphviz diagram showing the optimal installed capacity and commodity flows for the ‘utopia’ test system in 1990. Technologies are shown as boxes, commodities as circles, with arrows indicating energy flows. These diagrams are automatically generated when `graphviz_output = true` is set in the configuration file.

MATHEMATICAL FORMULATION

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

The heart of Temoa is a technology explicit energy system optimization model. It is an algebraic network of linked processes – where each process is defined by a set of engineering characteristics (e.g. capital cost, efficiency, capacity factor, emission rates) – that transform raw energy sources into end-use demands. The model objective function minimizes the present-value cost of energy supply by optimizing installed capacity and its utilization over time.

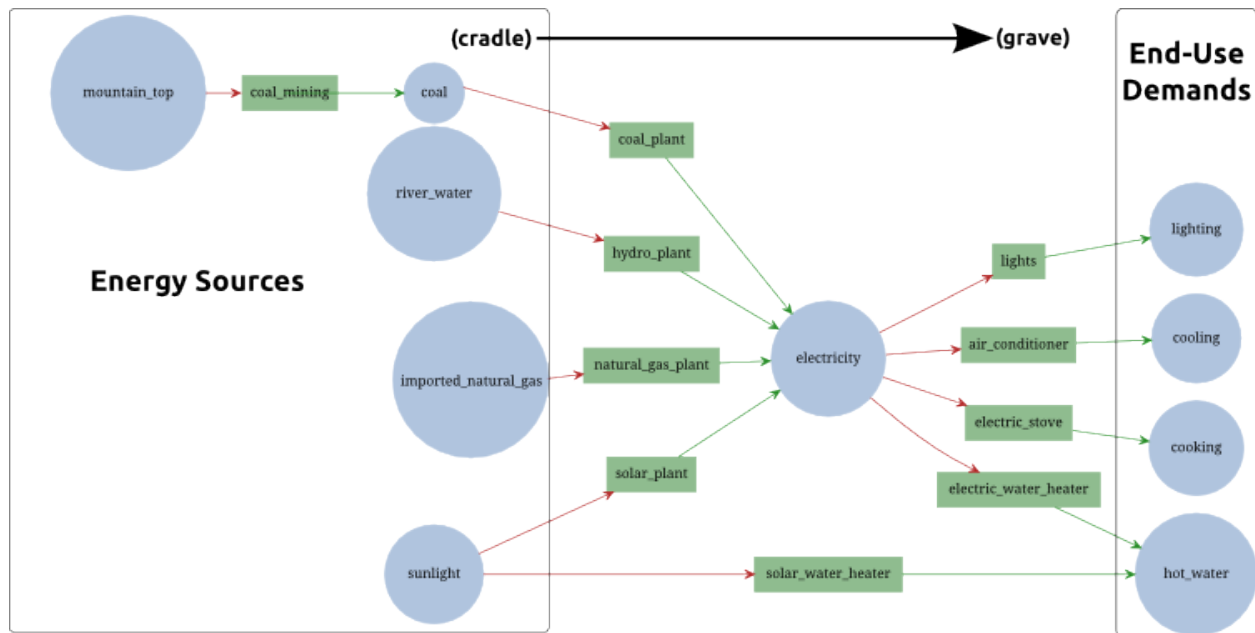
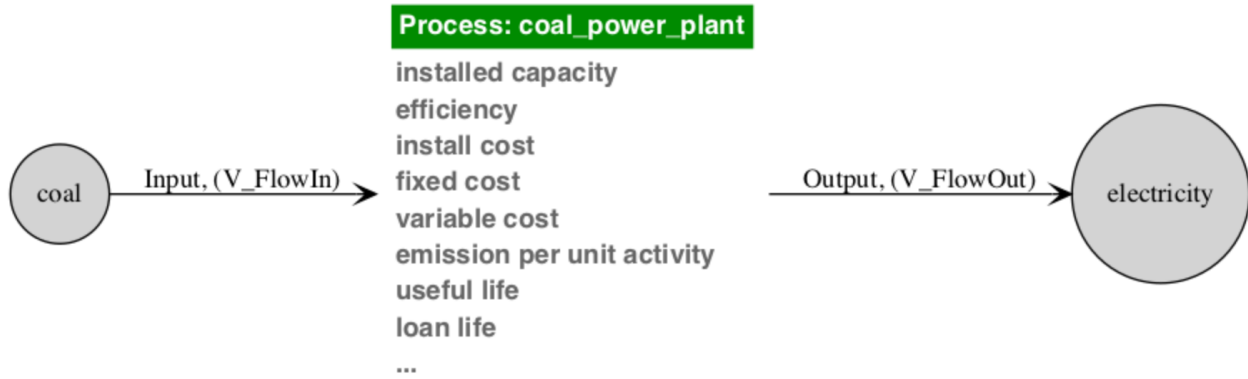


Fig. 1: A common visualization of energy system models is a directed network graph, with energy sources on the left and end-use demands on the right. The modeler must specify the end-use demands to be met, the technologies defined within the system (rectangles), and the inputs and outputs of each (red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

The most fundamental tenet of the model is the understanding of energy flow, treating all processes as black boxes that take inputs and produce outputs. Specifically, Temoa does not care about the inner workings of a process, only its global input and output characteristics. In this vein, the above graphic can be broken down into process-specific elements. For example, the coal power plant takes as input coal and produces electricity, and is subject to various costs (e.g. variable costs) and constraints (e.g. efficiency) along the way.

The modeler defines the processes and engineering characteristics through a combination of sets and parameters, described in the next few sections. Temoa then utilizes these parameters, along with the associated technology-specific decision



variables for capacity and activity, to create the objective function and constraints that are used during the optimization process.

7.1 Temoa Notation

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:

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

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

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

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

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

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

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

- We use the word ‘slice’ to refer to the tuple of season and time of day $\{s, d\}$. Note that these time slices are user-defined, and can represent time ranging large blocks of time (e.g., winter-night) to every hour in a given season.
- We use the word ‘process’ to refer to the tuple of technology and vintage $\{t, v\}$. For example, solar PV (technology) installed in 2030 (vintage).

7.2 Treatment of Time

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

- **Periods** - consecutive blocks of years, marked by the first year in the period. For example, a two-period model might consist of $P^f = \{2010, 2015, 2025\}$, representing the two periods of years from 2010 to 2015, and from 2015 to 2025. It is up to the model builder whether this represents start of year to start of year or end of year to end of year. Note that the last period element (2025) does not represent a new time period, but rather defines the end of the second time period and therefore the planning horizon.
- **Seasonal** - Each year is divided into one or more seasons. What a “season” represents depends on the `time_sequencing` mode chosen in the configuration file (see below). Each season carries a `segment_fraction` value in the `time_season` table specifying the fraction of the year it represents.
- **Daily** - Within each season, the day is subdivided into one or more time-of-day segments. Each segment has an associated `hours` value in the `time_of_day` table indicating how many hours it represents (e.g. 8 hours for “Day”, 16 hours for “Night”). Less detailed databases may use a single segment covering the full day.

We use the word ‘slice’ or ‘timeslice’ to refer to the tuple of season and time of day $\{s, d\}$. The fraction of a year represented by each slice is computed automatically:

$$SEG_{s,d} = \text{segment_fraction_per_season}(s) \times \frac{\text{hours}(d)}{\sum_{d'} \text{hours}(d')}$$

The sum of $SEG_{s,d}$ over all (s, d) pairs must equal 1.

7.2.1 Time Sequencing Modes

Temoa v4 supports four modes for interpreting the meaning and ordering of seasons, controlled by the `time_sequencing` setting in the configuration TOML file:

- **consecutive_days** — Each season represents a single day, and the seasons are treated as consecutive days in order. Use this when modeling e.g., a representative week (7 seasons) or a full year (365 seasons). Inter-season state (e.g. for storage) carries forward naturally from one season to the next, so the `time_season_sequential` table can be left empty (more on that below).
- **seasonal_timeslices** (*default*) — Each season represents a sequential slice of the year containing one or many days (e.g. Winter, Spring, Summer, Fall). The true chronological sequence is assumed to follow the `time_season` ordering, so the `time_season_sequential` table can be left empty. This is the traditional Temoa time representation.
- **representative_periods** — Each season represents a block of days that may not be contiguous in time (e.g. a “typical summer weekday” drawn from several months). If the model uses inter-season constraints such as seasonal storage or inter-season ramping, the true chronological sequence must be defined in the `time_season_sequential` table. Technologies using seasonal storage must also be flagged in the `technology` table.
- **manual** — The sequence of time slices is defined explicitly by the modeler in an optional `time_manual` table. This is an advanced feature provided in case none of the above modes are suitable, and is not recommended for most users.

The selected mode, in combination with the sequence of the `time_season` and `time_of_day` tables, is used to construct the `time_next` dictionary, which maps each (s, d) time slice to its successor in the chronological sequence, (s_{next}, d_{next}) .

The `days_per_period` configuration setting (default: 365) specifies how many days each planning period’s representative year encompasses. This is used to scale flow variables correctly. For example, reduce it to 7 when modeling a single representative week.

7.2.2 Time Season Sequential

When using the **representative_periods** time sequencing mode, the seasons in `time_season` may represent non-contiguous blocks of time (e.g. a “typical summer weekday” drawn from several calendar months). For constraints that depend on inter-season ordering — seasonal storage and inter-season ramping — the model needs to know the true chronological sequence showing how the representative seasons are stitched together to form a complete year.

The `time_season_sequential` table provides this mapping. Each row defines a *sequential season* (`seas_seq`) that references one of the `time_season` entries and carries its own `segment_fraction` and an integer `sequence` column that determines the chronological order. Because the same representative season may appear more than once in the reconstructed year (e.g. “typical summer weekday” could appear for June, July, and August), the sequential table can contain multiple entries that map back to the same `time_season` row, each with its own fraction.

From this table Temoa builds two internal dictionaries:

- `time_next_sequential` — maps each sequential season to its successor, forming a circular chain that represents the annual cycle.
- `sequential_to_season` — maps each sequential season back to its parent representative season in `time_season`.

These dictionaries are consumed by:

- The **seasonal storage energy constraint**, which chains the storage state of charge across sequential seasons in chronological order.
- The **ramp up/down season constraints**, which limit the rate of activity change at season boundaries that are adjacent in the sequential ordering but not necessarily adjacent in the `time_season` set.

For the **consecutive_days** and **seasonal_timeslices** modes, the `time_season_sequential` table may be left empty — the model derives the chronological order directly from the `time_season` set.

7.2.3 Periods

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

Temoa assumes that all elements of the `time_exist` 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.

One final note on periods: rather than optimizing each year within a period individually, Temoa makes the simplifying assumption that each time period contains n copies of a single, representative year. Temoa optimizes capacity and activity for just this characteristic year within each time period, assuming the results for different years in the same time period are identical. The Temoa objective function, however, accounts for the total cost across all years in all model time periods. Figure 3.3 gives a graphical explanation of the annual delineation.

As noted above, Temoa allows the modeler to subdivide each year into a set of time slices, comprised of a season and a time of day. Unlike `time_future`, there is no restriction on what labels the modeler may assign to the `time_season` and `time_of_day` set elements. Each season carries a `segment_fraction` (fraction of the year), and each time-of-day segment carries an `hours` value. These are combined to compute the `segment_fraction` for each (s, d) slice as described above.

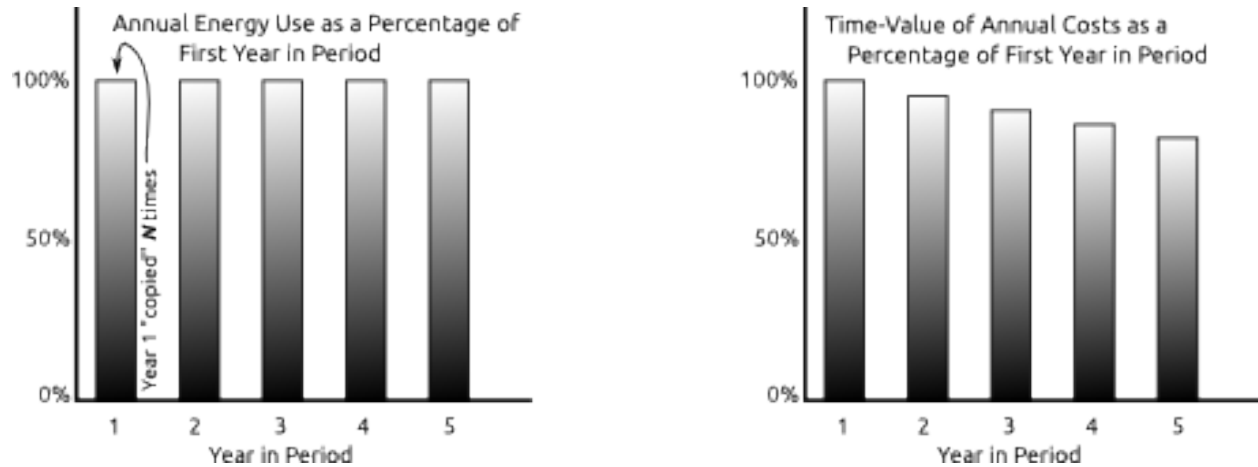


Fig. 2: The left graph is of energy, while the right graph is of the annual costs. The energy used in a period by a process is the same for all years (with exception for those processes that cease their useful life mid-period). However, even though the costs incurred will be the same, the time-value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

7.3 Sets

In a mathematical model like Temoa, sets are used to index parameters and variables. To enable the modeler to translate between the algebraic formulation, input database, and model code, we provide the mathematical notation used in the model formulation, as well as the associated names in the Python code and database schema in the tables below. The code representation is more verbose than the algebraic version, using full words.

Our discussion of sets is broken down into several logical categories to make it easier to parse. The **first group of sets pertains to Temoa's treatment of time**, as shown below. **Note:** Some entries in the "Database Table" column are comma-separated. In those cases, the first element refers to the name of the database table, and the second refers to specific column within the database table used to identify a specific subset. For example, in the first table below, `time_period` is the master set and `time_exist` is a subset that defines the user-defined model time periods to define historical technology vintages that exist prior to the model's time horizon for optimization.

Set	Database Table	Model Element	Notes
P^e	time_period, flag = e	time_exist	model periods prior to the beginning of the optimization time horizon
P^f	time_period, flag = f	time_future	model periods within the optimization time horizon
$*P^o$	time_period	time_optimize	model time periods to optimize, $(P^f - \max(P^f))$; the last time period is removed as it represents the end of the final period
$*V$	time_period	vintage_exist, vintage_optimize, vintage_all	tech vintages, $(P^e \cup P^o)$, derived from time period set and used to track technology vintages across time periods
D	time_of_day	time_of_day	intraday time divisions; each entry carries an hours value (default 1) specifying how many hours it represents
S	time_season	time_season	intra-annual divisions (e.g. seasons or representative days); each entry carries a segment_fraction giving its share of the year; ordered by sequence column
	time_season_sequence	time_season_sequence or- dered_season_sequence	superimposed sequential seasons used for seasonal storage and inter-season ramping; only required when time_sequencing = 'representative_periods'

The sets in the table below define Temoa's representation of **regions**.

Set	Database Table	Model Element	Notes
R	region	regions	distinct geographical regions
	region	regional_indices	set of all the possible combinations of interregional exchanges plus original region indices
		regional_global_indices	set of all used combinations of interregional exchanges plus original region indices and groups

The sets below define how technologies are represented within Temoa. Because technologies can serve many different functions across an energy system, we need to define a large number of **technology subsets**.

Set	Database Table	Model Element	Notes
*T	technology	tech_all	all technologies to be modeled (in v4, all technologies are production-type: $T = T^p$)
T^p	technology, flag LIKE 'p%'	tech_production	all production technologies, including baseload and storage ($T^b \cup T^s \subset T^p$)
T^b	technology, flag = pb	tech_baseload	baseload electric generators, which have constant output across intraday time segments ($T^b \subset T$)
T^s	technology, flag = ps	tech_storage	all storage technologies ($T^s \subset T$)
T^a	technology, annual = 1	tech_annual	technologies that produce constant annual output ($T^a \subset T$)
T^{res}	technology, reserve = 1	tech_reserve	electric generators contributing to the reserve margin requirement ($T^{res} \subset T$)
T^c	technology, curtail = 1	tech_curtailment	technologies with curtailable output and no upstream cost ($T^c \subset (T - T^{res})$)
T^f	technology, flex = 1	tech_flex	technologies producing excess commodity flows ($T^f \subset T$)
T^x	technology, exchange = 1	tech_exchange	technologies used for interregional commodity flows ($T^x \subset T$)
T^{ur}	ramp_up_hourly	tech_upramping	electric generators with a ramp up hourly rate limit; derived from ramp_up_hourly table ($T^{ur} \subset T$)
T^{dr}	ramp_down_hourly	tech_downramping	electric generators with a ramp down hourly rate limit; derived from ramp_down_hourly table ($T^{dr} \subset T$)
T^{ret}	technology, retire = 1	tech_retirement	technologies allowed to retire before end of life ($T^{ret} \subset (T - T^u)$)
T^u	technology, unlimited_cap = 1	tech_uncap	technologies that have no bound on capacity ($T^u \subset (T - T^{res})$)
T^{ss}	technology, flag = 'ps' AND seas_stor = 1	tech_seasonal_storage	seasonal storage technologies; requires both storage flag and seas_stor column ($T^{ss} \subset T^s$)
	tech_group	tech_group_names	named groups for use in group constraints
	tech_group_member	tech_group_members	each technology belonging to each group
T^e	existing_capacity	tech_exist	existing technologies with a past vintage ($T^e \subset T$)

The sets below define **commodities** that are consumed and produced by different energy technologies.

Set	Database Table	Model Element	Notes
C^d	commodity, flag = d	commodity_demand	end-use demand commodities, representing the final consumer demands
C^e	commodity, flag = e	commodity_emissions	emission commodities (e.g., CO ₂ NO _x); filtered by flag
C^p	commodity, flag IN (p, wp, s, a, wa)	commodity_physical	superset of physical, source, annual, and their waste variants; includes all non-demand, non-emission commodities
C^w	commodity, flag IN (w, wa, wp)	commodity_waste	commodity whose production can be greater than its consumption; can be physical, annual, or neither (not balanced, all wasted)
C^a	commodity, flag IN (a, wa)	commodity_annual	commodities whose flows are only balanced over each period, not per-timeslice ($C^a \subset C^p$)
$*C^l$		commodity_flex	derived set of commodities produced by a flex technology ($C^l \subset C^p$); auto-populated from tech_flex outputs
C^s	commodity, flag = s	commodity_source	primary source commodities, not balanced by CommodityBalance_constraint
$*C^c$		commodity_carrier	union of physical, demand, and waste commodities, ($C_p \cup C_d \cup C^w$)
$*C$		commodity_all	union of all commodity sets; union of carrier and emissions commodities

There is an additional set that defines **operators** (=, <, >). While not strictly necessary, defining these operators as a set allows modelers to express constraints more efficiently.

Set	Database Table	Model Element	Notes
	operator	operator	constraint operators

As indicated below, there are additional sets that are derived within the model code and thus do not appear in the database schema.

Set	Model Element	Notes
	tech_with_capacity	technologies eligible for capacitzation; computed as tech_all - tech_uncap
	tech_or_group	technologies or groups combined; union of tech_group_names tech_all
$*C^c$	commodity_carrier	physical energy carriers and end-use demands; union of physical, demand, and waste commodities
$*C$	commodity_all	union of all commodity sets; union of carrier and emissions commodities
T^e	tech_exist	technologies with existing capacity; derived from existing_capacity table

There are also python dictionaries and sets used to specify internal data structures during model construction and are not considered formal model elements:

- process_inputs, process_outputs, process_loans

- `active_flow_rpsditvo`, `active_flow_rpitvo`
- Various vintage and operational tracking dictionaries
- Time sequencing dictionaries (`time_next`, `time_next_sequential`)

7.4 Parameters

A summary table of input parameters is provided below, followed by a more detailed description of each.

Parameters are indexed by the sets defined above and used to specify input data. In the leftmost column below, the subscripts indicate the sets used to index the parameter. As with sets, we categorize parameters thematically and present them below in separate tables.

Below are the **time-related** parameters.

Parameter	Database Table	Model Element	Notes
SFS_s	<code>time_season</code>	<code>segment_fraction_per_season</code>	per-season year fraction; loaded from the <code>segment_fraction</code> column of <code>time_season</code>
	<code>time_of_day</code>	<code>time_of_day_hours</code>	hours per time-of-day segment; loaded from the <code>hours</code> column of <code>time_of_day</code> (default 1)
$SEG_{s,d}$	<i>(computed)</i>	<code>segment_fraction</code>	fraction of year represented by each (s, d) tuple; computed as <code>segment_fraction_per_season(s) × hours(d) / Σ hours</code>
	<i>(config)</i>	<code>time_sequencing</code>	time sequencing mode: <code>consecutive_days</code> , <code>seasonal_timeslices</code> (default), <code>representative_periods</code> , or <code>manual</code>
	<i>(config)</i>	<code>days_per_period</code>	number of days per period (default: 365); set as a top-level key in the configuration TOML file, not in the database
$SFS_{s^{seq}}$	<code>time_season_sequential</code>	<code>segment_fraction_per_season</code>	per-sequential-season year fraction; loaded from the <code>segment_fraction</code> column of <code>time_season_sequential</code>

Parameters in the table below relate to **capacity and its performance characteristics**.

Parameter	Database Table	Model Element	Notes
$C2A_{r,t}$	capacity_to_activity	capacity_to_activity	converts from capacity to activity units
$CC_{r,p,t,v}$	capacity_credit	capacity_credit	process-specific capacity credit used in the static reserve margin constraint
$CFT_{r,s,d,t}$	capacity_factor_tech	capacity_factor_tech	technology-specific capacity factor
$CFP_{r,s,d,t,v}$	capacity_factor_process	capacity_factor_process	process-specific capacity factor; allows capacity factor to change with technology vintage
$ECAP_{r,t,v}$	existing_capacity	existing_capacity	installed capacity that exists prior to first model time period
PRM_r	planning_reserve_margin	planning_reserve_margin	planning reserve margin used to ensure sufficient generating capacity
$RCD_{r,s,t,v}$	reserve_capacity_derate	reserve_capacity_derate	capacity derate factor for dynamic reserve margin constraint
$RUH_{r,t}$	ramp_up_hourly	ramp_up_hourly	hourly rate at which generation techs can ramp output up
$RDH_{r,t}$	ramp_down_hourly	ramp_down_hourly	hourly rate at which generation techs can ramp output down

Parameters in the table below relate to the specification of **costs**.

Parameter	Database Table	Model Element	Notes
$CF_{r,p,t,v}$	cost_fixed	cost_fixed	fixed operations & maintenance cost
$CI_{r,t,v}$	cost_invest	cost_invest	tech-specific investment cost
$CV_{r,p,t,v}$	cost_variable	cost_variable	variable operations & maintenance (O&M) cost
$CE_{r,p,e}$	cost_emission	cost_emission	emission costs
GDR	metadata_real	global_discount_rate	global rate used to convert future time period costs to the present cost
DLR	metadata_real	default_loan_rate	default loan rate used to amortize investment costs
$LLP_{r,t,v}$	loan_lifetime_process	loan_lifetime_process	process-specific loan term (default=lifetime_process)
$LR_{r,t,v}$	loan_rate	loan_rate	process-specific interest rate on investment cost

Parameters in the table below relate to the specification of **technology efficiency and performance**.

Parameter	Database Table	Model Element	Notes
$EFF_{r,i,t,v,o}$	efficiency	efficiency	technology- and commodity-specific conversion efficiency
$EFFV_{r,s,d,i,t,v,o}$	efficiency_variable	efficiency_variable	optional time slice multiplier on efficiency (no period index; applies to all periods)
$LTT_{r,t}$	lifetime_tech	lifetime_tech	technology-specific lifetime (default=40 years)
$LTP_{r,t,v}$	lifetime_process	lifetime_process	tech- and vintage-specific lifetime (default=lifetime_tech)
$LSC_{r,p,t,v}$	lifetime_survival_curve	lifetime_survival_curve	surviving fraction of original capacity
$SD_{r,t}$	storage_duration	storage_duration	storage duration per technology, specified in hours

Parameters in the table below relate to the specification of **end-use demands**.

Parameter	Database Table	Model Element	Notes
$DEM_{r,p,c}$	demand	demand	end-use demand by region-period-commodity
$DSD_{r,s,d,c}$	demand_specific_distribution	demand_specific_distribution	fractional annual demand by time slice (s,d)

Parameters in the table below relate to the specification of **emissions**.

Parameter	Database Table	Model Element	Notes
$EAC_{r,e,i,t,v,o}$	emission_activity	emission_activity	activity-based emissions rate
$EE_{r,t,v,e}$	emission_embodied	emission_embodied	emissions associated with the creation of capacity; embodied emissions
$EEOL_{r,t,v,e}$	emission_end_of_life	emission_end_of_life	emissions associated with the retirement/end of life of capacity

Parameters in the table below relate to the specification of **absolute limits on capacity, activity and emissions**.

Parameter	Database Table	Model Element	Notes
$LC_{r,p,t}$	limit_capacity	limit_capacity	limit on capacity by period; tech_or_group column accepts a technology name or group name
$LNC_{r,t,v}$	limit_new_capacity	limit_new_capacity	limit on new capacity deployment by vintage; tech_or_group column accepts a technology name or group name
$LA_{r,p,t}$	limit_activity	limit_activity	limit on activity by region and period; tech_or_group column accepts a technology name or group name
$LE_{r,p,e}$	limit_emission	limit_emission	limit on emissions by region and period
$LS_{r,t}$	limit_resource	limit_resource	cumulative activity limit across time periods (not supported in myopic); tech_or_group column accepts a technology name or group name

Parameters in the table below relate to the specification of **growth and degrowth limits**.

Parameter	Database Table	Model Element	Notes
$LGC_{r,t}$	limit_growth_capacity	limit_growth_capacity	capacity growth rate limits; tech_or_group column accepts a technology name or group name
$LDGC_{r,t}$	limit_degrowth_capacity	limit_degrowth_capacity	capacity degrowth rate limits; tech_or_group column accepts a technology name or group name
$LGNC_{r,t}$	limit_growth_new_capacity	limit_growth_new_capacity	new capacity growth rate limits; tech_or_group column accepts a technology name or group name
$LDGNC_{r,t}$	limit_degrowth_new_capacity	limit_degrowth_new_capacity	new capacity degrowth rate limits; tech_or_group column accepts a technology name or group name
$LGNC_{\Delta,r,t}$	limit_growth_new_capacity	limit_growth_new_capacity	new capacity growth acceleration limits; tech_or_group column accepts a technology name or group name
$LDGNC_{\Delta,r,t}$	limit_degrowth_new_capacity	limit_degrowth_new_capacity	new capacity degrowth deceleration limits; tech_or_group column accepts a technology name or group name

Parameters in the table below relate to the specification of **operational and split limits**.

Parameter	Database Table	Model Element	Notes
$LACF_{r,t,v,o}$	limit_annual_capacity	limit_annual_capacity	annual capacity factor limits; indexed by vintage (not period); tech_or_group column accepts a technology name or group name
$LSCF_{r,s,t}$	limit_seasonal_capacity	limit_seasonal_capacity	seasonal capacity factor limits (no period index; applies to all periods)
$LSF_{r,s,d,t}$	limit_storage_level	limit_storage_fraction	limit on storage level in any time slice
$TIS_{r,p,i,t}$	limit_tech_input	limit_tech_input_split	technology input split constraints specifying input fuel ratio at time slice level
$TISA_{r,p,i,t}$	limit_tech_input	limit_tech_input_split	technology input split constraints specifying input fuel ratio at average annual level
$TOS_{r,p,t,o}$	limit_tech_output	limit_tech_output_split	technology split constraints specifying output fuel ratio at time slice level
$TOSA_{r,p,t,o}$	limit_tech_output	limit_tech_output_split	technology split constraints specifying output fuel ratio at average annual level

Parameters in the table below relate to the specification of **share limits**.

Parameter	Database Table	Model Element	Notes
$LCS_{r,p,g1,g2}$	limit_capacity_share	limit_capacity_share	capacity share limits between technology groups
$LAS_{r,p,g1,g2}$	limit_activity_share	limit_activity_share	activity share limits between technology groups
$LNCS_{r,g1,g2,v}$	limit_new_capacity_share	limit_new_capacity_share	new capacity share limits

Parameters in the table below relate to the specification of **policy**.

Parameter	Database Table	Model Element	Notes
	rps_requirement	renewable_portfolio_standards	[Deprecated] RPS requirements; use limit_activity_share instead
$LIT_{r,t,e,t'}$	linked_tech	linked_techs	dummy techs used to convert CO2 emissions to physical commodity

Parameters in the table below relate to the specification of **construction and end-of-life**.

Parameter	Database Table	Model Element	Notes
$CON_{r,i,t,v}$	construction_input	construction_input	construction input requirements that represent commodities consumed by creation of process capacity
$EOLO_{r,t,v,o}$	end_of_life_output	end_of_life_output	end-of-life outputs that represent commodities produced by retirement/end of life of capacity

Parameters in the table below are **computed in code (i.e., model-derived)** and therefore do not appear in the database.

Parameter	Database Table	Model Element	Notes
* LEN_p		period_length	number of years in period p ; computed from time periods
* $LA_{t,v}$		loan_annualize	loan amortization by tech and vintage; based on DR_t ; computed from loan rate and life-time
* $PLF_{r,p,t,v}$		process_life_frac	fraction of available process capacity by region and period; computed from process life fraction

The tables below specify **model outputs**. When constructing a new database, they are left blank. These tables are auto-populated by the model after successful run completion.

- output_dual_variable
- output_objective
- output_curtailment
- output_net_capacity
- output_built_capacity
- output_retired_capacity
- output_flow_in
- output_flow_out
- output_flow_out_summary
- output_storage_level
- output_emission
- output_cost

Finally, the database tables below do not have a directed mapping to the model code.

Database Table	Purpose	Notes
myopic_efficiency	Myopic mode efficiency	alternative efficiency for myopic optimization
sector_label	Sectoral classification	used in output tables only

7.4.1 efficiency

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

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

7.4.2 efficiency_variable

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

When a technology's conversion efficiency varies by time of day or season — for example, a heat pump whose efficiency differs with temperature — the modeler can use `efficiency_variable` to specify these time-slice-dependent efficiency values. If not specified for a given process, it defaults to 1, meaning the base `efficiency` value applies uniformly. Note that there is no period index: the time-varying efficiency applies to all periods.

7.4.3 capacity_credit

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

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

7.4.4 capacity_factor_tech

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

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

7.4.5 capacity_factor_process

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

In addition to `capacity_credit`, 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, `capacity_factor_process` allows users to specify the capacity factor by season, time-of-day, technology, and vintage.

7.4.6 capacity_to_activity

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

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

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

or

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

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?"

7.4.7 cost_fixed

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

The `cost_fixed` 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 to 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 ($\frac{\text{Dollars}}{\text{UnitCap}}$).

7.4.8 cost_invest

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

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

7.4.9 cost_variable

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

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

7.4.10 cost_emission

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

The `cost_emission` parameter specifies a cost per unit of emission for a given emissions commodity in each period. This allows the modeler to penalize emission production, for example via a carbon tax. The cost is applied to total emissions of commodity e in period p .

7.4.11 construction_input

$$CON_{r \in R, i \in C^p, t \in T \setminus T^u, v \in V}$$

The `construction_input` parameter allows the modeller to attach commodity input flows to the production of new capacity, in units of activity per unit capacity. Assumes that capacity is produced evenly over years in its vintage period.

7.4.12 demand

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

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

To specify a non-uniform distribution of demand across time slices, use the `demand_specific_distribution` (DSD) parameter, described next.

7.4.13 demand_specific_distribution

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

If there is an end-use demand that varies over the course of a day or across seasons – for example, heating or cooling in the summer or winter – the modeler may specify the fraction of annual demand occurring in each time slice. The sum of DSD for each demand commodity c must be 1. If the modeler does not define DSD for a demand commodity, Temoa automatically populates it using the `segment_fraction` values, resulting in a uniform (flat) distribution proportional to the length of each time slice.

7.4.14 emission_activity

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

Temoa currently has two methods for enabling a process to produce an output: the `efficiency` parameter, and the `emission_activity` parameter. Where the `efficiency` parameter defines the amount of output energy a process produces per unit of input, the `emission_activity` 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).

7.4.15 emission_embodied

$$EE_{r \in R, t \in T \setminus T^u, v \in V, e \in C_e}$$

Like the `emission_activity` parameter, but attaches emission outputs to the creation of capacity instead of activity flows. Assumes that capacity is produced evenly over each year in the deployment vintage.

7.4.16 emission_end_of_life

$$EEO_{r \in R, t \in T \setminus T^u, v \in V, e \in C_e}$$

Like `emission_embodied`, but attaches emissions to the retirement/end of life of capacity rather than production of capacity. Assumes that retirement or end of life occur evenly over years in that period.

7.4.17 end_of_life_output

$$EOLO_{r \in R, t \in T \setminus T^u, v \in V, o \in C_p}$$

Like `construction_input`, but attaches flows to the retirement/end of life of capacity rather than production of capacity. Assumes that retirement or end of life occur evenly over years in that period.

7.4.18 existing_capacity

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

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

7.4.19 global_discount_rate

$$GDR$$

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

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

where n is in years. This parameter is used to calculate all discounted costs, which are the basis of the objective function. Costs are discounted to the first future time period in the model.

The output in the `output_cost` table shows both discounted and non-discounted (raw) values for all model costs.

The `global_discount_rate` is entered in the `metadata_real` table in the database.

7.4.20 loan_lifetime_process

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

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

7.4.21 lifetime_process

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

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

7.4.22 lifetime_survival_curve

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

The `lifetime_survival_curve` parameter represents the surviving fraction of original capacity for a given process at a given period. It allows the modeler to specify gradual capacity loss over time rather than abrupt retirement at end of life. Values should be between 0 and 1, where 1 means full survival.

7.4.23 lifetime_tech

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

Similar to `lifetime_process`, this parameter specifies the total useful life of a given technology in years, with a default of 40 years. If all vintages of a given technology have the same lifetime, then `lifetime_tech` can be used instead of `lifetime_process`.

7.4.24 linked_techs

$$LIT_{r \in R, t \in T, e \in C^e, t' \in T}$$

In power-to-gas pathways, `CO2` is an input to some processes, including synthetic natural gas production and liquid fuel production via Fischer-Tropsch. Within the model, `CO2` must be converted from an emissions commodity to a physical commodity that can be included in the `efficiency` table. The `linked_techs` parameter specifies the dummy technology used to convert an emissions commodity to a physical commodity. Note that the first `t` represents the primary upstream technology linked to the dummy linked technology, which is represented by the second `t'` index.

7.4.25 loan_rate

$$LR_{r \in R, t \in T, v \in V}$$

The interest rate used for loans supporting investment costs. The default loan rate is accessible in the `metadata_real` table in the database.

7.4.26 default_loan_rate

$$DLR$$

A scalar parameter specifying the default interest rate applied to loans when no process-specific `loan_rate` is defined. This value is read from the `metadata_real` table in the database.

7.4.27 limit_activity

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

The `limit_activity` parameter places an upper or lower bound on the total activity (energy production) from a technology or technology group in each model time period. The bound direction is controlled by the `operator` column (le, ge, or eq). The `tech_or_group` column accepts either a single technology name or a group name defined in `tech_group_names`.

7.4.28 limit_activity_share

$$LAS_{r \in R, p \in P, g_1, g_2}$$

The `limit_activity_share` parameter constrains the activity of one technology or group as a share of another technology or group's activity. For example, it can enforce a minimum renewable generation share. The `operator` column controls whether the share is an upper bound, lower bound, or equality.

7.4.29 limit_annual_capacity_factor

$$LACF_{r \in R, t \in T, v \in V, o \in C_c}$$

The `limit_annual_capacity_factor` parameter limits the annual capacity factor of a process — that is, the ratio of actual annual output to maximum possible output. The `tech_or_group` column accepts a technology name or group name. Note this is indexed by vintage, not period, and will be applied to all valid time periods for that process.

7.4.30 limit_capacity

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

The `limit_capacity` parameter places an upper or lower bound on the total available (retirement-adjusted) capacity of a technology or technology group in each model timeperiod. The `operator` column controls the bound direction. The `tech_or_group` column accepts either a single technology name or a group name.

7.4.31 limit_capacity_share

$$LCS_{r \in R, p \in P, g_1, g_2}$$

The `limit_capacity_share` parameter constrains the capacity of one technology group as a share of another group's capacity. The `operator` column controls the bound direction. The `group` columns accept either a single technology name or a technology group name.

7.4.32 limit_degrowth_capacity

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

The `limit_degrowth_capacity` parameter defines the maximum annual rate at which the total capacity of a technology (or group) can shrink between periods. The `tech_or_group` column accepts a technology name or group name.

7.4.33 limit_degrowth_new_capacity

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

The `limit_degrowth_new_capacity` parameter constrains the rate of decrease in new capacity deployment between consecutive periods.

7.4.34 limit_degrowth_new_capacity_delta

$$LDGNC_{\Delta, r \in R, t \in T}$$

The `limit_degrowth_new_capacity_delta` parameter constrains the deceleration of new capacity degrowth between periods, essentially adding “inertia” to the degrowth of new capacity deployment.

7.4.35 limit_emission

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

The `limit_emission` parameter ensures that Temoa finds a solution that fits within the modeler-specified limit on emission e in time period p . The operator column controls whether this is an upper bound, lower bound, or equality.

7.4.36 limit_growth_capacity

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

The `limit_growth_capacity` parameter defines the maximum annual rate at which the total capacity of a technology (or group) can grow between periods. The `tech_or_group` column accepts a technology name or group name.

7.4.37 limit_growth_new_capacity

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

The `limit_growth_new_capacity` parameter constrains the rate of increase in new capacity deployment between consecutive periods.

7.4.38 limit_growth_new_capacity_delta

$$LGNC_{\Delta, r \in R, t \in T}$$

The `limit_growth_new_capacity_delta` parameter constrains the acceleration of new capacity growth between periods. This essentially adds “inertia” to the growth of new capacity deployment.

7.4.39 limit_new_capacity

$$LNC_{r \in R, t \in T, v \in V}$$

The `limit_new_capacity` parameter constrains the amount of new capacity that can be deployed for a given technology or group in a specific vintage. The `tech_or_group` column accepts a technology name or group name.

7.4.40 limit_new_capacity_share

$$LNCS_{r \in R, g_1, g_2, v \in V}$$

The `limit_new_capacity_share` parameter constrains the new capacity of one technology or group as a share of another technology or group’s new capacity.

7.4.41 limit_resource

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

The `limit_resource` parameter represents a bound on the cumulative amount of commodity that can be produced by a technology or group over the entire model time horizon. The `tech_or_group` column accepts a technology name or group name. Note that this is *not* supported in myopic mode as the cumulative limit would need to decline as the horizon moves forward, which has not been added yet.

7.4.42 limit_seasonal_capacity_factor

$$LSCF_{r \in R, s \in S, t \in T}$$

The `limit_seasonal_capacity_factor` parameter limits the capacity factor of a technology in a specific season. There is no period index: the limit applies across all periods. The `tech_or_group` column accepts a technology name or group name.

7.4.43 limit_storage_fraction

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

The `limit_storage_fraction` parameter constrains the storage level of a storage technology in any time slice to be at or above a specified fraction of its maximum charge. Values should be between 0 and 1. Note that this constraint will be applied to all valid period and vintage combinations for the technology.

7.4.44 limit_tech_input_split

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

Some technologies have a single output but have multiple input fuels. The `limit_tech_input_split` parameter constrains the shares of commodity input to a specific technology in a given period in every time slice. The `operator` column controls whether the share is an upper bound, lower bound, or equality.

7.4.45 limit_tech_input_split_annual

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

Similar to `limit_tech_input_split`, but constrains the average input commodity shares at the annual level, allowing the shares at the time slice level to vary.

7.4.46 limit_tech_output_split

$$TOS_{r \in R, p \in P, t \in T, o \in C_o}$$

Some technologies have a single input fuel but have multiple outputs. The `limit_tech_output_split` parameter constrains the shares of commodity output from a specific technology in a given period by time slice.

7.4.47 limit_tech_output_split_annual

$$TOSA_{r \in R, p \in P, t \in T, o \in C_o}$$

Similar to `limit_tech_output_split`, but constrains the average output commodity shares at the annual level, allowing the shares at the time slice level to vary.

7.4.48 planning_reserve_margin

$$PRM_{r \in R}$$

The `planning_reserve_margin` parameter specifies the capacity reserve margin in the electric sector by region. The capacity reserve margin represents the installed generating capacity — expressed as a share of peak load — that must be available in reserve to meet contingencies. Temoa estimates peak demand from electricity production by time slice.

7.4.49 ramp_down_hourly

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

The `ramp_down_hourly` parameter specifies the fraction of installed capacity by which a technology can ramp output down per hour. This is used in the `ramp_down_day_constraint` (between time-of-day slices) and `ramp_down_season_constraint` (between the last time-of-day slice in one season and the first in the next).

7.4.50 ramp_up_hourly

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

The `ramp_up_hourly` parameter specifies the fraction of installed capacity by which a technology can ramp output up per hour. This is used in the `ramp_up_day_constraint` (between time-of-day slices) and `ramp_up_season_constraint` (between seasons).

7.4.51 reserve_capacity_derate

$$RCD_{r \in R, s \in S, t \in T^{res}, v \in V}$$

The `reserve_capacity_derate` parameter allows the modeler to derate the capacity of a reserve technology in specific seasons — for example, to account for seasonal availability. Values default to 1 (no derate). This parameter is used in the ‘dynamic’ version of the `reserve_margin` constraint.

7.4.52 segment_fraction

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

The `segment_fraction` parameter specifies the fraction of the year represented by each combination of season and time of day. In v4, this parameter is **computed automatically** from two user-specified inputs:

- `segment_fraction` column in the `time_season` table — the fraction of the year each season represents (e.g. 0.25 for each quarter).
- `hours` column in the `time_of_day` table — the number of hours each time-of-day segment represents (e.g. 8 for “Day”, 16 for “Night”).

The computation is:

$$SEG_{s,d} = \text{segment_fraction_per_season}(s) \times \frac{\text{hours}(d)}{\sum_{d'} \text{hours}(d')}$$

The sum of all $SEG_{s,d}$ values must equal 1, representing 100% of a year.

7.4.53 segment_fraction_per_season

$$SFS_{s \in S}$$

The per-season share of the year, loaded directly from the `segment_fraction` column of the `time_season` database table. These values are the first factor in the *segment_fraction* computation and must sum to 1.

7.4.54 segment_fraction_per_sequential_season

$$SFS_{s^{seq} \in S^{seq}}$$

The per-sequential-season share of the year, loaded from the `segment_fraction` column of the `time_season_sequential` database table. Used to compute the `days_adjust` ratio that rescales storage levels and flows between non-sequential and sequential season representations.

7.4.55 storage_duration

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

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

7.4.56 *loan_annualize

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

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

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

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

7.4.57 *period_length

$$LEN_{p \in P}$$

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,

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

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

7.4.58 *process_life_frac

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

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

long through the period the process is active. This parameter is that ratio, formally defined as:

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

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

Note that this parameter is defined over the same indices as `cost_variable` – 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 $TLF_{2012,car,2010} = \frac{3}{8}$.

7.5 Decision Variables

Decision variables are the quantities optimized by the model. A summary table of decision variables is provided below, followed by a more detailed description of each.

Table 1: Temoa's Main Variables

Variable	Temoa Name	Short Description
$FO_{r,p,s,d,i,t,v,o}$	<code>v_flow_out</code>	Commodity flow by time slice out of a tech based on a given input
$FOA_{r,p,i,t,v,o}$	<code>v_flow_out_annual</code>	Annual commodity flow out of a tech based on a given input
$FIS_{r,p,s,d,i,t,v,o}$	<code>v_flow_in</code>	Commodity flow into a storage tech to produce a given output
$FLX_{r,p,s,d,i,t,v,o}$	<code>v_flex</code>	The portion of commodity production exceeding demand
$FLXA_{r,p,i,t,v,o}$	<code>v_flex_annual</code>	The portion of commodity production from constant production techs exceeding demand
$CUR_{r,p,s,d,i,t,v,o}$	<code>v_curtailment</code>	Commodity flow out of a tech that is curtailed
$CAP_{r,p,t,v}$	<code>v_capacity</code>	Required tech capacity to support associated activity
$CAPAVL_{r,p,t}$	<code>v_capacity_available</code>	Derived variable representing the capacity of technology t available in period p
$SI_{r,p,s,t,v}$	<code>v_storage_init</code>	Hub variable for the initial charge level of each daily storage cycle
$SL_{r,p,s,d,t,v}$	<code>v_storage_level</code>	Charge level each time slice associated with storage techs
$SSL_{r,p,s,t,v}$	<code>v_seasonal_storage_l</code>	Base charge level of sequential seasons for seasonal storage
$RCAP_{r,p,t,v}$	<code>v_retired_capacity</code>	Capacity retired before end of life
$ART_{r,p,t,v}$	<code>v_annual_retirement</code>	Annualised capacity retiring or reaching end of life
$NCAP_{r,t,v}$	<code>v_new_capacity</code>	New deployed capacity

7.5.1 v_flow_out

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

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

7.5.2 v_flow_out_annual

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

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

7.5.3 v_flex

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

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

7.5.4 v_flex_annual

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

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

7.5.5 v_curtailment

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

The `v_curtailment` variable is an accounting tool to help calculate the unused production capacity of technologies annotated in the Technology database table as curtailable technologies belonging to the `tech_curtailment` set. Renewables such as wind and solar are often placed in this set. While we used to simply formulate the `Capacity` and `CommodityBalance` constraints as inequalities that implicitly allowed for curtailment, this simpler approach does not work with renewable targets because the curtailed portion of the electricity production counts towards the target, and there is no way to distinguish it from the useful production. Including an explicit curtailment term addresses the issue. Curtailment in the model is simply the production activity that is not used in the model and is reported as such in the `output_curtailment` table. Note: Outputs presented in the `output_curtailment` table for curtailment (the table separately includes flex outputs) are limited by Capacity Factor. Meaning: if a tech has a capacity of 10 units, and a CF of 0.8 and a usage of 5 units, then the reported curtailment is 3 units ($0.8 \times 10 - 5$).

7.5.6 v_flow_in_storage

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

Because the production and consumption associated with storage techs occur across different time slices, the commodity flow into a storage technology cannot be discerned from `v_flow_out`. Thus an explicit `flow_in` variable is required for storage.

7.5.7 v_capacity

$$CAP_{r,p,t,v}$$

The `v_capacity` variable represents the available capacity of a process (r, t, v) in period p , adjusted for retirements or end of life. Temoa constrains the capacity variable to be able to meet the total commodity flow out of that process in all time slices in which it is active (7.1).

7.5.8 v_capacity_available_by_period_and_tech

$CAPAVL_{r,p,t}$

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

7.5.9 v_storage_init

$SI_{r,p,s,t,v}$

The `v_storage_init` variable replaces the `v_storage_level` variable for the initial storage charge level of each season. This is purely for solver optimisation and has no impact on model structure or results. This 1:1 swap measurably improves presolve time in large models with storage. The mechanism behind this is not understood.

7.5.10 v_storage_level

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

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

7.5.11 v_seasonal_storage_level

$SSL_{r,p,s,t,v}$

The `v_seasonal_storage_level` variable tracks the base charge level at the boundary between sequential seasons for technologies flagged as seasonal storage (`tech_seasonal_storage`). It is used in the `seasonal_storage_energy_constraint` to chain the storage state of charge across seasons defined in the `time_season_sequential` table.

7.5.12 v_retired_capacity

$RCAP_{r,p,t,v}$

The `v_retired_capacity` variable tracks the cumulative capacity of a process that has been retired before its natural end of life, up to and including period p . Only technologies in the `tech_retirement` set may retire early in this manner. This variable is non-decreasing across periods.

7.5.13 v_annual_retirement

$ART_{r,p,t,v}$

The `v_annual_retirement` variable represents the annualised capacity that retires or reaches end of life in period p . It accounts for both early retirements (via `v_retired_capacity`) and natural end-of-life.

7.5.14 v_new_capacity

$NCAP_{r,t,v}$

The `v_new_capacity` variable represents the newly deployed capacity of a technology in its vintage period. It is the primary investment decision variable and drives the capital cost terms in the objective function. Unlike `v_capacity`, it has no period index — capacity is built once in its vintage year.

We explain the equations governing these variables in the [Equations](#) section.

7.6 Equations

There are four main equations that govern the flow of energy through the model network. The `Demand_Constraint` (7.6) ensures that the supply meets demand in every time slice. For each process, the `Capacity_constraint` (7.1) ensures that there is sufficient capacity to meet the optimal commodity flows across all time slices. Between processes, the `CommodityBalance_constraint` (7.7) ensures that global commodity production across the energy system is sufficient to meet the endogenous demands for that commodity. Finally, the objective function (7.24) drives the model to minimize the system-wide cost of energy supply by optimizing the deployment and utilization of energy technologies across the system.

One additional point regarding the model formulation. Technologies that produce constant annual output can be placed in the `tech_annual` set. While not required, doing so improves computational performance by eliminating the season and time of day (s, d) indices associated with these technologies. In order to ensure the model functions correctly with these simplified technologies, slightly different formulations of the capacity and commodity balance constraints are required. See the `AnnualCommodityBalance_constraint` and `CapacityAnnual_constraint` (7.2) below for details.

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

A couple notes on the notation below. First, in all equations, we **bold** variables to distinguish them from parameters. Second, you will notice the use of the Theta superset (Θ). The Temoa code makes heavy use of sparse sets, for both correctness and efficient use of computational resources. For brevity, and to avoid discussion of implementation details, we do not enumerate their logical creation here. Instead, we rely on the reader's general understanding of the context. The use of (Θ) means that the constraint is only defined for the exact indices that the modeler specified.

7.6.1 Capacity-Defining Constraints

We begin with the `Capacity_constraint` and `CapacityAnnual_constraint`, which are particularly important because they define the relationship between installed capacity and allowable commodity flow.

`temoa.components.capacity.capacity_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

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

$$(\text{CFP}_{r,s,d,t,v} \cdot \text{C2A}_{r,t} \cdot \text{SEG}_{s,d}) \cdot \text{CAP}_{r,p,t,v} = \sum_{I,O} \text{FO}_{r,p,s,d,i,t,v,o} + \sum_{I,O} \text{CUR}_{r,p,s,d,i,t,v,o} \quad (7.1)$$

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

`temoa.components.capacity.capacity_annual_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

Similar to `Capacity_constraint`, but for technologies belonging to the `tech_annual` set. Technologies in the `tech_annual` set have constant output across different timeslices within a year, so we do not need to ensure that installed capacity is sufficient across all timeslices, thus saving some computational effort. Instead, annual output is sufficient to calculate capacity. Hourly capacity factors cannot be defined to annual technologies but annual

capacity factors can be set using `limit_annual_capacity_factor`, which will be implicitly accounted for here.

$$C2A_{r,t} \cdot \mathbf{CAP}_{r,p,t,v} \geq \sum_{I,O} \mathbf{FOA}_{r,p,i,t \in T^a,v,o} \quad (7.2)$$

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

`temoa.components.capacity.adjusted_capacity_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

This constraint updates the capacity of a process by taking into account retirements and end of life. For a given (r, p, t, v) index, this constraint sets the capacity equal to the amount installed in period v and subtracts from it any and all retirements that occurred prior to the period in question, p , and end of life from the survival curve if defined. It finally adjusts for the process life fraction, which accounts for a possible mid-period end of life where, for example, EOL 3 years into a 5-year period would be treated as $\frac{3}{5}$ capacity for all 5 years.

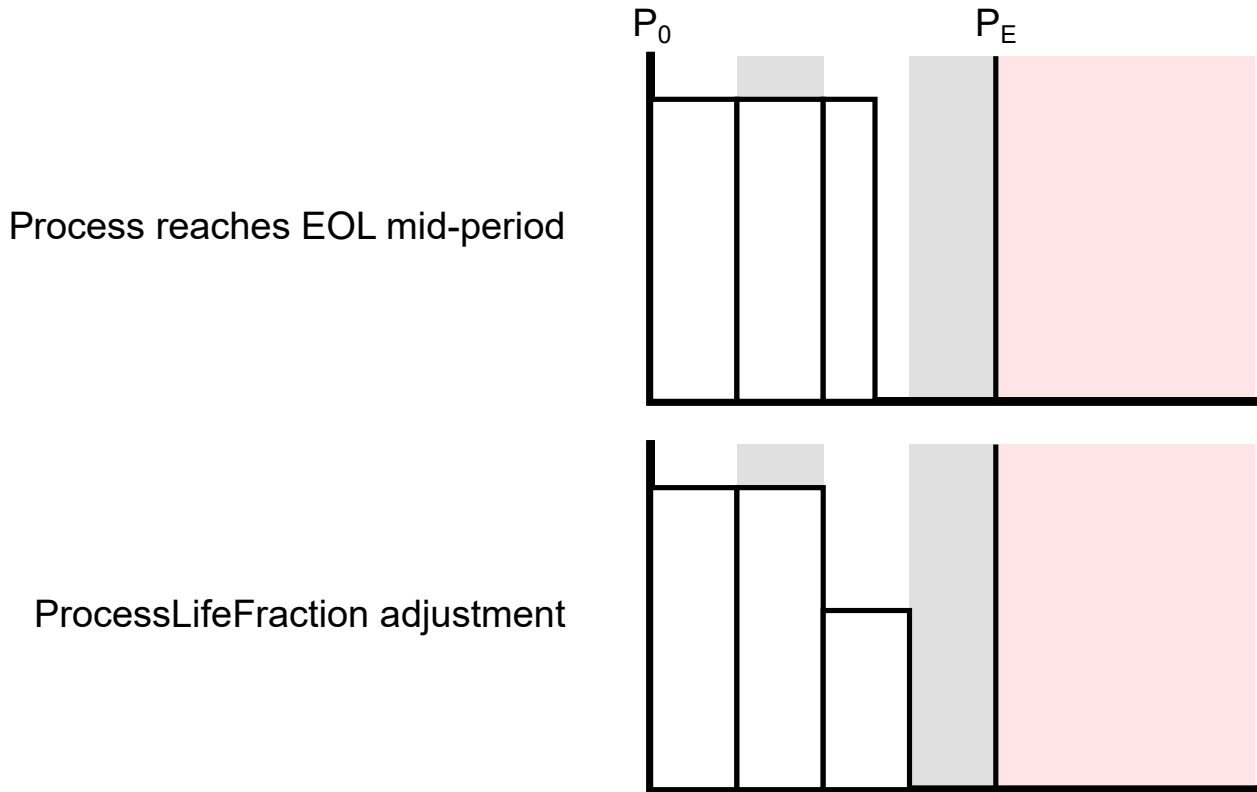


Fig. 3: For processes reaching end of life mid-period, the process life fraction adjustment is applied, distributing the effective capacity over the whole period.

For processes using survival curves, the yearly survival curve $LSC_{r,p,t,v}$ is averaged over the period to get the effective remaining capacity for that period. Because this implicitly handles mid-period end of life, $PLF_{r,p,t,v}$ is used to account for both phenomena.

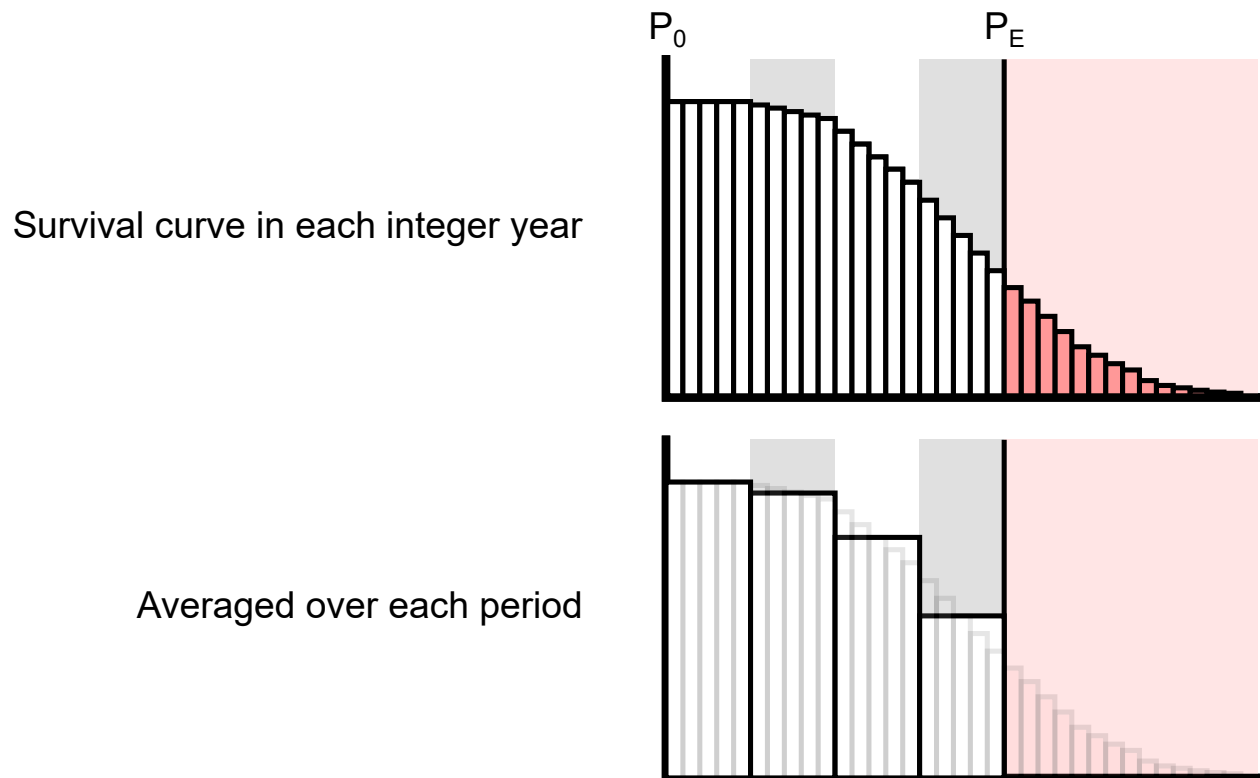


Fig. 4: For processes with a defined survival curve, the surviving capacity is averaged over each period to get the adjusted capacity. This implicitly handles mid-period end of life as a survival curve will always be zero after the end of life of a process.

$$\mathbf{CAP}_{r,p,t,v} = \begin{cases} \text{PLF}_{r,p,t,v} \cdot \left(\text{ECAP}_{r,t,v} - \sum_{v < p' \leq p} \frac{\mathbf{RCAP}_{r,p',t,v}}{\text{LSC}_{r,p',t,v}} \right) & \text{if } v \in T^e \\ \text{PLF}_{r,p,t,v} \cdot \left(\mathbf{NCAP}_{r,t,v} - \sum_{v < p' \leq p} \frac{\mathbf{RCAP}_{r,p',t,v}}{\text{LSC}_{r,p',t,v}} \right) & \text{if } v \notin T^e \end{cases} \quad (7.3)$$

$$\text{where } \text{PLF}_{r,p,t,v} = \begin{cases} \frac{1}{\text{LEN}_p} \cdot \left(\sum_{y=p}^{p+\text{LEN}_p-1} \text{LSC}_{r,y,t,v} \right) & \text{if } t \in T^{sc} \\ \frac{1}{\text{LEN}_p} \cdot (v + \text{LTP}_{r,t,v} - p) & \text{if } t \notin T^{sc} \end{cases}$$

We divide $\frac{\mathbf{RCAP}_{r,p',t,v}}{\text{LSC}_{r,p',t,v}}$ because the average survival factor in $\text{PLF}_{r,p,t,v}$ is indexed to the vintage period (the beginning of the survival curve). So, we adjust for the relative survival from the time when that retirement occurred (treated here as at the beginning of each period).

`temoa.components.capacity.annual_retirement_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

Get the annualised retirement rate for a process in a given period. Used to output retirement (including end of life, EOL) and to model end of life flows and emissions. Assumes that retirement from the beginning of each period is evenly distributed over that model period $\frac{1}{\text{LEN}_p}$ for the accounting of retirement flows (in the same way we assume capacity is deployed evenly over the model period for construction inputs and embodied emissions). The factor $\frac{\text{LSC}_{r,p,t,v}}{\text{PLF}_{r,p,t,v}}$ adjusts the average survival during a period to the survival at the beginning of that period.

$$\mathbf{ART}_{r,p,t,v} = \begin{cases} \frac{1}{\text{LEN}_p} \cdot \frac{\text{LSC}_{r,p,t,v}}{\text{PLF}_{r,p,t,v}} \cdot \mathbf{CAP}_{r,p,t,v} & \text{if EOL} \\ \frac{1}{\text{LEN}_p} \cdot \left(\frac{\text{LSC}_{r,p,t,v}}{\text{PLF}_{r,p,t,v}} \cdot \mathbf{CAP}_{r,p,t,v} - \frac{\text{LSC}_{r,p_{next},t,v}}{\text{PLF}_{r,p_{next},t,v}} \cdot \mathbf{CAP}_{r,p_{next},t,v} \right) & \text{otherwise} \end{cases} \quad (7.4)$$

where EOL when $p \leq v + \text{LTP}_{r,t,v} < p + \text{LEN}_p$

`temoa.components.capacity.capacity_available_by_period_and_tech_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology*) $\rightarrow$ ExprLike

The **CAPAVL** variable is nominally for reporting solution values, but is also used in the Limit constraint calculations.

$$\mathbf{CAPAVL}_{r,p,t} = \sum_{v, p_i \leq p} \mathbf{CAP}_{r,p,t,v} \quad (7.5)$$

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

7.6.2 Network Constraints

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

`temoa.components.commodities.demand_constraint` (*model: TemoaModel, r: Region, p: Period, dem: Commodity*) $\rightarrow$ ExprLike

The Demand constraint drives the model. This constraint ensures that supply meets the demand specified by the

Demand parameter in all periods, by ensuring that the sum of all the annual demand output commodity (*dem*) generated by **FOA** across all technologies and vintages equals the modeler-specified demand.

$$\sum_{I,T,V} \mathbf{FOA}_{r,p,i,t,v,dem} = DEM_{r,p,dem} \quad (7.6)$$

The per-timeslice distribution of demand across non-annual technologies is enforced by the separate `demand_activity_constraint`. In the case of a singleton demand, where only one (*r*, *i*, *t*, *v*) sub-process produces the demand commodity, the flow variables are fixed directly and demand constraints are skipped.

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

`temoa.components.commodities.commodity_balance_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, c: Commodity*) → ExprLike

Where the Demand constraint (7.6) ensures that end-use demands are met, the CommodityBalance constraint ensures that the endogenous system demands are met. This constraint requires the total production of a given commodity to equal the amount consumed, thus ensuring an energy balance at the system level. In this most general form of the constraint, the energy commodity being balanced has variable production at the time slice level. The energy commodity can then be consumed by three types of processes: storage technologies, non-storage technologies with output that varies at the time slice level, and non-storage technologies with constant annual output.

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

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

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

Consumption of the commodity by construction inputs is annualised using the period length. Production of the commodity by end-of-life outputs uses the AnnualRetirement variable, which is already annualised.

Finally, for annual commodities, AnnualCommodityBalance is used which balances the sum of flows over each year.

process outputs + imports + end of life outputs = process inputs + construction inputs + exports + flex waste

$$\begin{aligned}
& \sum_{I, t \notin T^a, V} \mathbf{FO}_{r,p,s,d,i,t,v,c} && \text{(processes outputting commodity)} \\
& + SEG_{s,d} \cdot \sum_{I, t \in T^a, V} \mathbf{FOA}_{r,p,i,t,v,c} && \text{(annual processes outputting commodity)} \\
& + \sum_{\text{reg} \neq r, I, t \in T^x, V} \mathbf{FIM}_{r-\text{reg},p,s,d,i,t,v,c} && \text{(inter-regional imports of commodity)} \\
& + SEG_{s,d} \sum_{T,V} (EOLO_{r,t,v,c} \cdot \mathbf{ART}_{r,p,t,v}) && \text{(end-of-life outputs of commodity)} \\
& \left\{ \begin{array}{l} = \text{if } c \notin C^w \\ \geq \text{if } c \in C^w \end{array} \right. \\
& \sum_{t \in T^s, V, O} \mathbf{FIS}_{r,p,s,d,c,t,v,o} && \text{(commodity stored)} \\
& + \sum_{t \notin T^s, V, O} \frac{\mathbf{FO}_{r,p,s,d,c,t,v,o}}{EFF_{r,c,t,v,o}} && \text{(commodity consumed by processes)} \\
& + SEG_{s,d} \cdot \sum_{t \in T^a, V, O} \frac{\mathbf{FOA}_{r,p,c,t,v,o}}{EFF_{r,c,t,v,o}} && \text{(commodity consumed by annual processes)} \\
& + \sum_{\text{reg} \neq r, t \in T^x, V, O} \mathbf{FEX}_{r-\text{reg},p,s,d,c,t,v,o} && \text{(inter-regional exports of commodity)} \\
& + \sum_{I, t \in T^f, V} \mathbf{FLX}_{r,p,s,d,i,t,v,c} && \text{(flex wastes of commodity)} \\
& + SEG_{s,d} \cdot \sum_{T,V} \left(CON_{r,c,t,v} \cdot \frac{\mathbf{NCAP}_{r,t,v}}{LEN_p} \right) && \text{(consumed annually by construction inputs)} \\
& \forall \{r, p, s, d, c\} \in \Theta_{\text{CommodityBalance}}
\end{aligned} \tag{7.7}$$

`temoa.components.commodities.annual_commodity_balance_constraint` (*model: TemoaModel, r: Region, p: Period, c: Commodity*) → ExprLike

Similar to `CommodityBalance_constraint` but only balances the supply and demand of the commodity at the period level, summing all flows over the period but allowing imbalances at the time slice or seasonal level. Applies only to commodities in the `commodity_annual` set.

7.6.3 Physical and Operational Constraints

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

`temoa.components.operations.baseload_diurnal_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) → ExprLike

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

Ideally, this constraint would not be necessary, and baseload processes would simply not have a *d* index. However,

implementing the more efficient functionality is currently on the Temoa TODO list.

$$SEG_{s,D_0} \cdot \sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} = SEG_{s,d} \cdot \sum_{I,O} \mathbf{FO}_{r,p,s,D_0,i,t,v,o} \quad (7.8)$$

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

`temoa.components.commodities.demand_activity_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage, dem: Commodity*) $\rightarrow$ ExprLike

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

This constraint ensures that the ratio of a process activity to demand is constant for all time slices. In other words, while the model may choose how each technology contributes to a demand at the annual level, the time slice level distribution of the technology's contribution to the demand is fixed to the demand specific distribution.

$$\sum_I \mathbf{FOA}_{r,p,i,t,v,dem} \cdot \mathbf{DSD}_{r,s,d,dem} = \sum_I \mathbf{FO}_{r,p,s,d,i,t,v,dem} \quad (7.9)$$

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

Note that this constraint is unnecessary and therefore skipped for annual technologies or singleton demands.

`temoa.components.storage.storage_energy_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

This constraint enforces the continuity of storage level between time slices.

Uses the `time_next` mapping to chain storage levels forward. For non-seasonal storage, `v_storage_init` replaces `v_storage_level` on the LHS at the daily cycle wrap point (last time-of-day). Combined with the tie-back constraint ($SL_{d_{last}} = SI$), this is structurally equivalent to a closed cycle.

Empirically, this swap improved barrier solve time ~25% on a 16-region national model, though the structural reason is not fully understood.

Non-seasonal, last time-of-day (`d_last`):

$$SI_{r,p,s,t,v} + \text{net_charge} = SL_{r,p,s_{next},d_{next},t,v}$$

All other time slices (non-seasonal and seasonal):

$$SL_{r,p,s,d,t,v} + \text{net_charge} = SL_{r,p,s_{next},d_{next},t,v}$$

For seasonal storage, the last time-of-day is skipped (handled by `SeasonalStorageEnergy_constraint`).

`temoa.components.storage.seasonal_storage_energy_constraint` (*model: TemoaModel, r: Region, p: Period, s_seq: Season, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

This constraint enforces the continuity of state of charge between seasons for seasonal storage. Sequential season storage level increases by the matched season's net charge over that entire day, adjusted for number of days represented by sequential vs non-sequential seasons. Only applies to storage technologies in the

tech_seasonal_storage set. s^* represents the matching non-sequential season for the sequential season s^{seq} .

$$\begin{aligned} \text{SSL}_{r,p,s^{seq},t,v} + \frac{\text{SFS}_{s^{seq}}}{\text{SFS}_{s^*}} \cdot \left(\text{SL}_{r,p,s^*,d_{last},t,v} + \sum_{I,O} \text{FI}_{r,p,s^*,d_{last},i,t,v,o} \cdot \text{EFF}_{r,i,t,v,o} - \sum_{I,O} \text{FO}_{r,p,s^*,d_{last},i,t,v,o} \right) \\ = \frac{\text{SFS}_{s^{seq}}}{\text{SFS}_{s^*}} \cdot \text{SL}_{r,p,s_{next},d_{first},t,v} + \text{SSL}_{r,p,s_{next},t,v} \end{aligned} \quad (7.10)$$

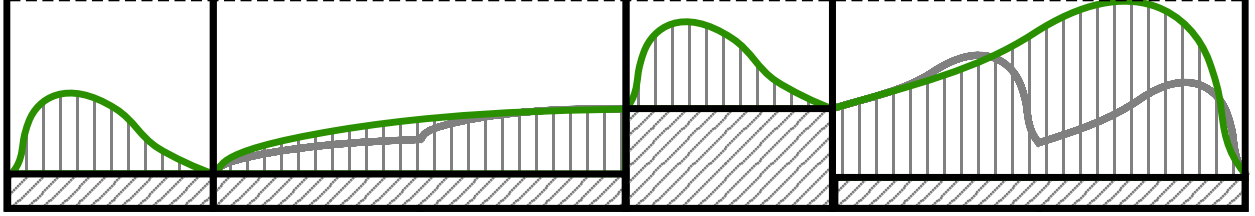


Fig. 5: How sequential seasons chain together for seasonal storage. Hatched area is seasonal_storage_level $\text{SSL}_{r,p,s^{seq},t,v}$. Vertical lines are StorageLevel $\text{SL}_{r,p,s^*,d,t,v}$. Green line is net seasonal storage level $\text{SSL}_{r,p,s^{seq},t,v} + \text{SL}_{r,p,s^*,d,t,v}$. Background grey lines show how storage levels from non-sequential seasons are combined in sequential seasons. Dashed line is SeasonalStorageEnergyUpperBound. Sequential seasons two and four here are each two days while one and three are each one day.

temoa.components.storage.storage_energy_upper_bound_constraint (model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage) → ExprLike

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

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

$$\text{SL}_{r,p,s,d,t,v} \leq \text{CAP}_{r,p,t,v} \cdot C2A_{r,t} \cdot \frac{SD_{r,t}}{24 \cdot DPP} \cdot \sum_d \text{SEG}_{s,d} \cdot DPP \quad (7.11)$$

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

A season can represent many days. Within each season, flows are multiplied by the number of days each season represents and, so, the upper bound needs to be adjusted to allow day-scale flows (e.g., charge in the morning, discharge in the afternoon).

temoa.components.storage.seasonal_storage_energy_upper_bound_constraint (model: TemoaModel, r: Region, p: Period, s_seq: Season, d: TimeOfDay, t: Technology, v: Vintage) → ExprLike

Daily (non-seasonal) storage

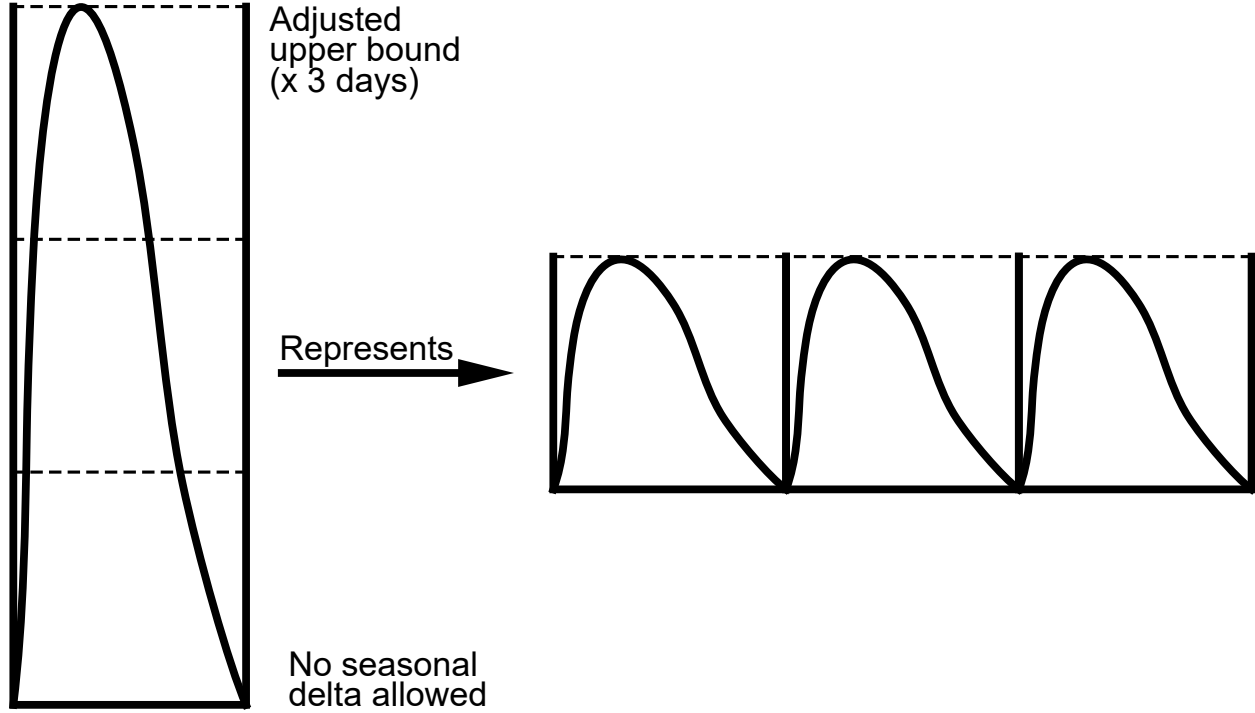


Fig. 6: Representation of a 3-day season for non-seasonal (daily) storage.

Builds off of `StorageEnergyUpperBound_constraint`. Enforces the max charge capacity of seasonal storage, summing the real storage level with the superimposed sequential seasonal storage level. s^* represents the matching non-sequential season for the sequential season s^{seq} .

$$\mathbf{SSL}_{r,p,s^{seq},t,v} + \mathbf{SL}_{r,p,s^*,d,t,v} \cdot \frac{\mathbf{SFS}_{s^{seq}}}{\mathbf{SFS}_{s^*}} \leq \mathbf{CAP}_{r,p,t,v} \cdot C2A_{r,t} \cdot \frac{SD_{r,t}}{24 \cdot DPP} \quad (7.12)$$

Unlike non-seasonal (daily) storage, seasonal storage is allowed to carry energy between seasons. However, through seasons representing multiple days, many days' charge deltas have accumulated, multiplied by the number of days the season represents. If we allowed these stacked deltas to carry between seasons then we would be multiplying the effective energy capacity of the storage. We could just constrain the seasonal delta to the unadjusted energy capacity, but then the final day in the season would sit atop a season's worth of deltas, possibly exceeding our upper or lower bound by a factor of $\frac{N-1}{N}$ where N is the number of days the sequential season represents.

So, we do not adjust the upper energy bound for seasonal storage. This limits the ability of seasonal storage to perform arbitrage within each season, but allows it to carry energy between seasons realistically.

`temoa.components.storage.storage_charge_rate_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

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

$$\sum_{I,O} \mathbf{FIS}_{r,p,s,d,i,t,v,o} \cdot \mathbf{EFF}_{r,i,t,v,o} \leq \mathbf{CAP}_{r,p,t,v} \cdot C2A_{r,t} \cdot \mathbf{SEG}_{s,d} \quad (7.13)$$

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

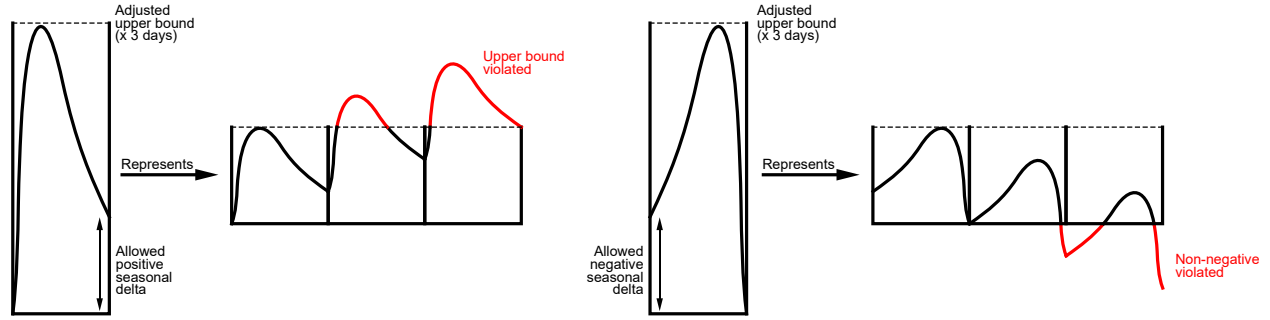


Fig. 7: The energy upper bound or non-negative lower bound could be violated in a season representing multiple days if we both adjusted the upper bound to the number of days and allowed a seasonal delta.

Seasonal storage

No adjustment

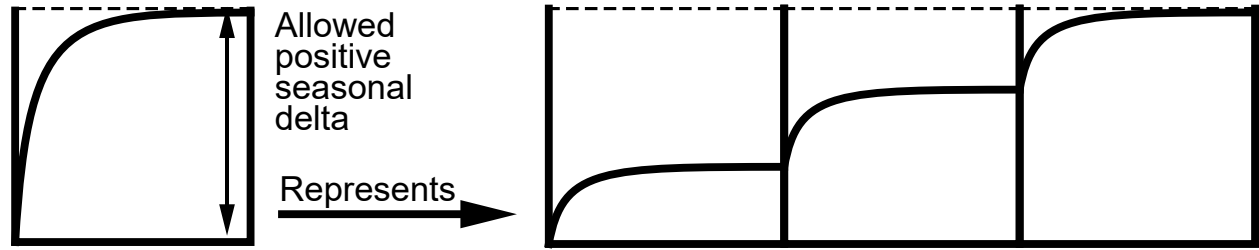


Fig. 8: Unadjusted energy upper bound constraint for seasonal storage.

`temoa.components.storage.storage_discharge_rate_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) → ExprLike

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

$$\sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \leq \mathbf{CAP}_{r,p,t,v} \cdot C2A_{r,t} \cdot SEG_{s,d} \quad (7.14)$$

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

`temoa.components.storage.storage_throughput_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) → ExprLike

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

$$\sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{I,O} \mathbf{FIS}_{r,p,s,d,i,t,v,o} \cdot EFF_{r,i,t,v,o} \leq \mathbf{CAP}_{r,p,t,v} \cdot C2A_{r,t} \cdot SEG_{s,d} \quad (7.15)$$

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

`temoa.components.operations.ramp_up_day_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) → ExprLike

One of two constraints built from the `ramp_up_hourly` table, along with the `ramp_up_season_constraint`. `ramp_up_day` constrains ramp rates between time slices within each season and `ramp_up_season` constrains ramp rates between sequential seasons. If the `time_sequencing` parameter is set to `consecutive_days` then the `ramp_up_season` constraint is skipped as seasons already connect together.

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. This constraint is only applied to technologies in the set `tech_upramping`. We assume for simplicity the rate limits do not vary with technology vintage. The ramp rate limits for a technology should be expressed in percentage of its rated capacity per hour.

In a representative periods or seasonal time slices model, the next time slice, (s_{next}, d_{next}) , from the end of each season, (s, d_{last}) is the beginning of the same season, (s, d_{first})

$$\frac{\sum_{I,O} \mathbf{FO}_{r,p,s_{next},d_{next},i,t,v,o}}{SEG_{s_{next},d_{next}} \cdot 24 \cdot DPP} - \frac{\sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o}}{SEG_{s,d} \cdot 24 \cdot DPP} \leq RUH_{r,t} \cdot \Delta H \cdot \frac{CAP_{r,p,t,v} \cdot C2A_{r,t}}{24 \cdot DPP} \quad (7.16)$$

$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{ramp_up_day}}$
where: $\Delta H = \frac{H_d + H_{d_{next}}}{2}$

where:

- $SEG_{s,d}$ is the fraction of the period in time slice (s, d)
- DPP is the number of days in each period
- $RUH_{r,t}$ is the ramp up rate per hour
- ΔH is the average of the hours in timeslice d and d_{next} , i.e. $(H_d + H_{d_{next}})/2$
- $CAP \cdot C2A / (24 \cdot DPP)$ gives the maximum hourly capacity

`temoa.components.operations.ramp_down_day_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

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

$$\frac{\sum_{I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o}}{SEG_{s,d} \cdot 24 \cdot DPP} - \frac{\sum_{I,O} \mathbf{FO}_{r,p,s_{next},d_{next},i,t,v,o}}{SEG_{s_{next},d_{next}} \cdot 24 \cdot DPP} \leq RDH_{r,t} \cdot \Delta H \cdot \frac{CAP_{r,p,t,v} \cdot C2A_{r,t}}{24 \cdot DPP} \quad (7.17)$$

$\forall \{r, p, s, d, t, v\} \in \Theta_{\text{ramp_down_day}}$
where: $\Delta H = \frac{H_d + H_{d_{next}}}{2}$

`temoa.components.operations.ramp_up_season_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, s_next: Season, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

Constrains the ramp up rate of activity between time slices at the boundary of sequential seasons. Same as `ramp_up_day` but only applies to the boundary between sequential seasons, i.e., (s^{seq}, d_{last}) to $(s_{next}^{seq}, d_{first})$ and s_{next}^{seq} is based on the `TimeSequential` table rather than the `time_season` table.

`temoa.components.operations.ramp_down_season_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, s_next: Season, t: Technology, v: Vintage*) $\rightarrow$ ExprLike

Constrains the ramp down rate of activity between time slices at the boundary of sequential seasons. Same as `ramp_down_day` but only applies to the boundary between sequential seasons, i.e., (s^{seq}, d_{last}) to $(s_{next}^{seq}, d_{first})$ and s_{next}^{seq} is based on the `TimeSequential` table rather than the `time_season` table.

`temoa.components.reserves.reserve_margin_static` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay*) $\rightarrow$ ExprLike

During each period p , the sum of capacity values of all reserve technologies $\sum_{t \in T^{res}} \mathbf{CAP}_{r,p,t,v}$, which are defined in the set $\mathbf{T}^{res}$, should exceed the peak load by PRM , 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. Each generator is allowed to contribute its available capacity times a pre-defined capacity credit, $CC_{r,p,t,v}$.

For exchange technologies (i.e., inter-regional transmission), reserve contributions are added to the downstream region but *subtracted* from the upstream region. This is because, since they are not generating any power, their summed contribution across regions should be zero.

$$\begin{aligned}
& \sum_{t \in T^{res} \setminus T^x, V} CC_{r,p,t,v} \cdot \mathbf{CAP}_{r,p,t,v} \cdot SEG_{s^*,d^*} \cdot C2A_{r,t} \\
& + \sum_{t \in T^{res} \cap T^x, V} CC_{r_i-r,p,t,v} \cdot \mathbf{CAP}_{r_i-r,p,t,v} \cdot SEG_{s^*,d^*} \cdot C2A_{r_i-r,t} \\
& - \sum_{t \in T^{res} \cap T^x, V} CC_{r-r_i,p,t,v} \cdot \mathbf{CAP}_{r-r_i,p,t,v} \cdot SEG_{s^*,d^*} \cdot C2A_{r-r_i,t} \\
& \geq \left[\sum_{t \in T^{res} \setminus T^x \setminus T^a, V, I, O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \right. \\
& + \sum_{t \in T^{res} \cap T^a, V, I, O} \begin{cases} DSD_{r,s,d,o} & \text{if } o \in C^d \\ SEG_{s,d} & \text{otherwise} \end{cases} \cdot \mathbf{FOA}_{r,p,i,t,v,o} \\
& + \sum_{t \in T^{res} \cap T^x, V, I, O} \mathbf{FO}_{r_i-r,p,s,d,i,t,v,o} \\
& - \sum_{t \in T^{res} \cap T^x, V, I, O} \mathbf{FI}_{r-r_i,p,s,d,i,t,v,o} \\
& \left. - \sum_{t \in T^{res} \cap T^s, V, I, O} \mathbf{FI}_{r,p,s,d,i,t,v,o} \right] \cdot (1 + PRM_r)
\end{aligned} \tag{7.18}$$

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

`temoa.components.reserves.reserve_margin_dynamic` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay*) $\rightarrow$ ExprLike

A dynamic alternative to the traditional, static reserve margin constraint. Capacity values are calculated from availability of generation in each hour—like an operating reserve margin—accounting for a capacity derate factor

subtracting, for example, forced outage due to icing.

$$\begin{aligned}
& \sum_{t \in T^{res} \setminus T^x \setminus T^s, V} CFP_{r,s^*,d^*,t,v} \cdot RCD_{r,s^*,t,v} \cdot \mathbf{CAP}_{r,p,t,v} \cdot SEG_{s^*,d^*} \cdot C2A_{r,t} \\
& + \sum_{t \in T^{res} \cap T^x \setminus T^s, V} CFP_{r_i-r,s^*,d^*,t,v} \cdot RCD_{r_i-r,s^*,t,v} \cdot \mathbf{CAP}_{r_i-r,p,t,v} \cdot SEG_{s^*,d^*} \cdot C2A_{r_i-r,t} \\
& - \sum_{t \in T^{res} \cap T^x \setminus T^s, V} CFP_{r-r_i,s^*,d^*,t,v} \cdot RCD_{r-r_i,s^*,t,v} \cdot \mathbf{CAP}_{r-r_i,p,t,v} \cdot SEG_{s^*,d^*} \cdot C2A_{r-r_i,t} \\
& + \sum_{t \in (T^s \cap T^{res}), V, I, O} (\mathbf{FO}_{r,p,s,d,i,t,v,o} - \mathbf{FI}_{r,p,s,d,i,t,v,o}) \cdot RCD_{r,s,t,v} \\
& \geq \left[\sum_{t \in T^{res} \setminus T^x \setminus T^a, V, I, O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \right. \\
& + \sum_{t \in T^{res} \cap T^a, V, I, O} \begin{cases} DSD_{r,s,d,o} & \text{if } o \in C^d \\ SEG_{s,d} & \text{otherwise} \end{cases} \cdot \mathbf{FOA}_{r,p,i,t,v,o} \\
& + \sum_{t \in T^{res} \cap T^x, V, I, O} \mathbf{FO}_{r_i-r,p,s,d,i,t,v,o} \\
& - \sum_{t \in T^{res} \cap T^x, V, I, O} \mathbf{FI}_{r-r_i,p,s,d,i,t,v,o} \\
& \left. - \sum_{t \in T^{res} \cap T^s, V, I, O} \mathbf{FI}_{r,p,s,d,i,t,v,o} \right] \cdot (1 + PRM_r)
\end{aligned} \tag{7.19}$$

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

`temoa.components.emissions.linked_emissions_tech_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage, e: Commodity*) $\rightarrow$ ExprLike

This constraint can be used for carbon capture technologies that produce CO2 as an emissions commodity, but the CO2 also serves as a physical input commodity to a downstream process, such as synthetic fuel production. To accomplish this, a dummy technology is linked to the CO2-producing technology, converting the emissions activity into a physical commodity amount as follows:

$$\begin{aligned}
& - \sum_{I, O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \cdot EAC_{r,e,i,t,v,o} = \sum_{I, O} \mathbf{FO}_{r,p,s,d,i,\hat{t},v,o} \\
& \forall \{r, p, s, d, t, v, e\} \in \Theta_{\text{linked_techs}}
\end{aligned} \tag{7.20}$$

where $\hat{t} = LT_{r,t,e}$ is the linked technology for primary technology t and emissions commodity e . For annual technologies ($t \in T^a$ or $\hat{t} \in T^a$), the corresponding $\mathbf{FOA}$ variable scaled by DSD (for end use demand techs) or SEG (for all other annual techs) is used instead.

The relationship between the primary and linked technologies is given in the `linked_techs` table. It is implicit that the primary region corresponds to the linked technology as well. The lifetimes of the primary and linked technologies should be specified and identical.

7.6.4 Objective Function

`temoa.components.costs.annuity_to_pv` (*rate: float, periods: float*) → float | Expression

Multiplication factor to convert an annuity to net present value

$$\frac{P}{A}(i, N) = \frac{(1+i)^N - 1}{i(1+i)^N} \quad (7.21)$$

where:

- i is the interest/discount rate
- N is the number of periods

`temoa.components.costs.pv_to_annuity` (*rate: float, periods: float*) → float | Expression

Multiplication factor to convert net present value to an annuity

$$\frac{A}{P}(i, N) = \frac{i + (1+i)^N}{(1+i)^N - 1} \quad (7.22)$$

`temoa.components.costs.fv_to_pv` (*rate: float, periods: float*) → float | Expression

Multiplication factor to convert a future value to net present value

$$\frac{P}{F}(i, N) = \frac{1}{(1+i)^N} \quad (7.23)$$

`temoa.components.costs.total_cost_rule` (*model: TemoaModel*) → Expression

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

$$\begin{aligned} C_{loans} = & \sum_{r,t,v \in \Theta_{CI}} CI_{r,t,v} \cdot \mathbf{NCAP}_{r,t,v} && \text{(overnight capital cost)} \\ & \cdot \frac{A}{P}(i = \mathbf{LR}_{r,t,v}, N = \mathbf{LLP}_{r,t,v}) && \text{(overnight cost amortised into annual loan payments)} \\ & \cdot \frac{P}{A}(i = \mathbf{GDR}, N = \mathbf{LLP}_{r,t,v}) && \text{(annual loan payments discounted to NPV in vintage year)} \\ & \cdot \frac{A}{P}(i = \mathbf{GDR}, N = \mathbf{LTP}_{r,t,v}) && \text{(NPV reamortised over lifetime of process using GDR)} \\ & \cdot \frac{P}{A}(i = \mathbf{GDR}, N = \min(\mathbf{LTP}_{r,t,v}, P_e - v)) && \text{(costs within planning horizon discounted to NPV in vintage year)} \\ & \cdot \frac{P}{F}(i = \mathbf{GDR}, N = v - P_0) && \text{(NPV in vintage year discounted to base year } P_0) \end{aligned} \quad (7.24)$$

Note that capital costs ($CI_{r,t,v}$) are handled in several steps.

1. Each capital cost is amortized using the loan rate (i.e., technology-specific discount rate) and loan period.
2. The annual stream of payments is converted into a lump sum using the global discount rate and loan period.
3. The new lump sum is amortized at the global discount rate over the process lifetime.
4. Loan payments beyond the model time horizon are removed and the lump sum recalculated.
5. Finally, the lump sum is discounted back to the beginning of the horizon (P_0) using the global discount rate.

Steps 3 and 4 serve to correctly balance the cost-benefit of technologies whose useful lives would extend beyond the planning horizon. While an explicit salvage term is not included, this approach properly captures the capital costs incurred within the model time horizon, accounting for process-specific loan rates and periods.

In the case of processes using survival curves, steps 3 and 4 do not reamortise costs uniformly over the process lifetime. Instead, costs are amortised over the life of the process in proportion to the survival fraction in each year. Note that, for this calculation, a survival curve $LSC_{r,y,t,v}$ must be defined out to the year in which the surviving fraction is zero, even if that extends beyond the planning horizon. It must also be defined for each integer year between model periods and, if not, these gaps will be filled by linear interpolation ahead of this calculation.

$$\begin{aligned}
 C_{loans,LSC} = & \sum_{r,t,v \in \Theta_{CI}} CI_{r,t,v} \cdot \mathbf{NCAP}_{r,t,v} && \text{(overnight capital cost)} \\
 & \cdot \frac{A}{P}(i = \mathbf{LR}_{r,t,v}, N = \mathbf{LLP}_{r,t,v}) && \text{(overnight cost amortised into annual loan payments)} \\
 & \cdot \frac{P}{A}(i = \mathbf{GDR}, N = \mathbf{LLP}_{r,t,v}) && \text{(annual loan payments discounted to NPV in vintage year)} \\
 & \cdot \left(\sum_{v < Y} LSC_{r,y,t,v} \cdot \frac{P}{F}(i = \mathbf{GDR}, N = P - v + 1) \right)^{-1} && \text{(reamortised over survival curve (normalized))} \\
 & \cdot \sum_{v < Y < P_e} LSC_{r,y,t,v} \cdot \frac{P}{F}(i = \mathbf{GDR}, N = P - v + 1) && \text{(costs within planning horizon discounted to NPV in vintage year)} \\
 & \cdot \frac{P}{F}(i = \mathbf{GDR}, N = v - P_0) && \text{(NPV in vintage year discounted to base year } P_0)
 \end{aligned}
 \tag{7.25}$$

Where Y is the set of each integer year y within the planning horizon.

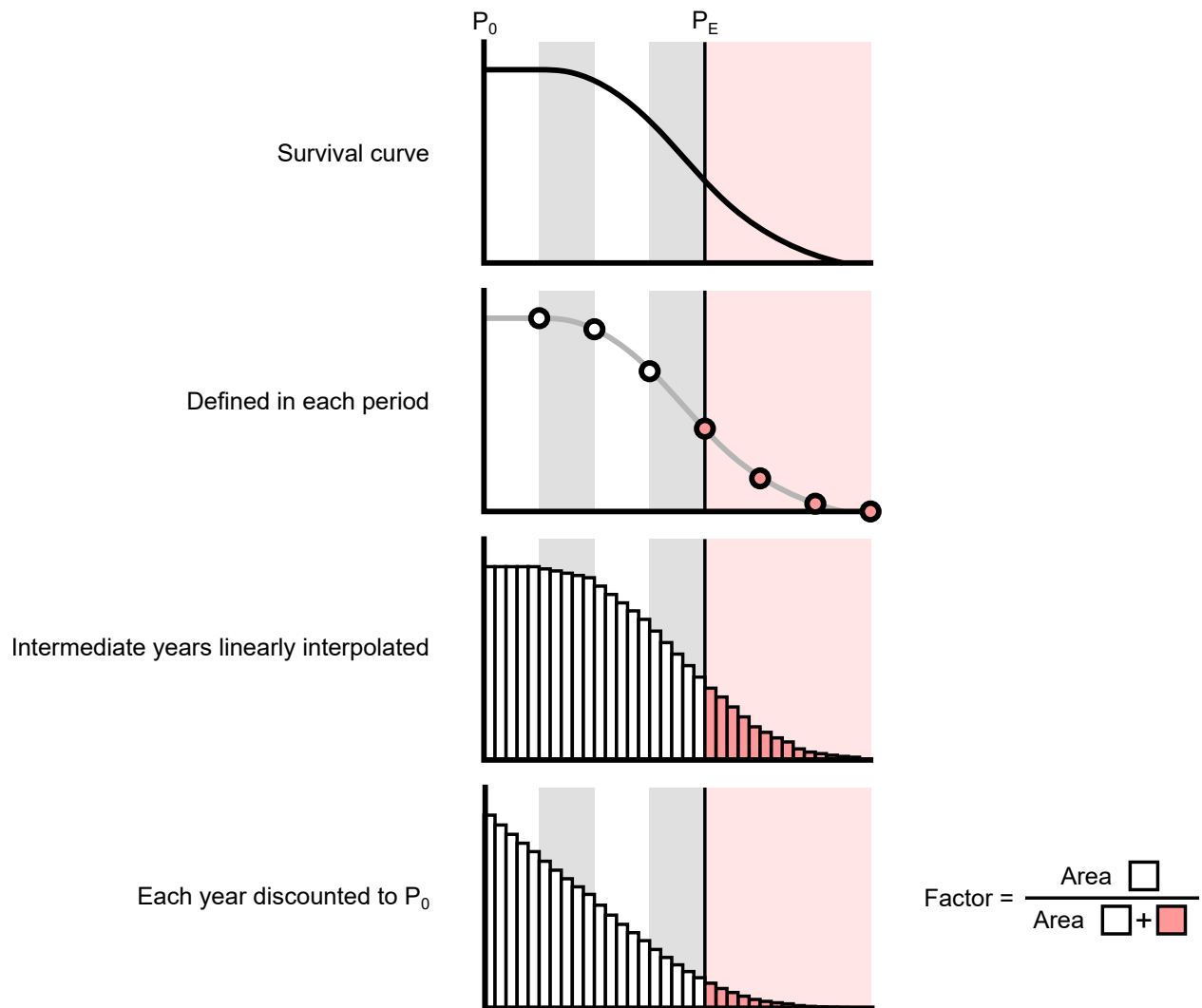


Fig. 9: Steps 3 and 4 for processes with survival curves.

Fixed, variable, and emissions annual cost factors are determined by:

$$\begin{aligned}
C_{fixed} &= \sum_{r,p,t,v \in \Theta_{CF}} CF_{r,p,t,v} \cdot \mathbf{CAP}_{r,p,t,v} && \text{(annual fixed cost)} \\
C_{variable} &= \sum_{r,p,t \notin T^a, v \in \Theta_{CV}} CV_{r,p,t,v} \cdot \sum_{S,D,I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} && \text{(annual variable cost on flow)} \\
&\quad \text{where } t \notin T^a \\
&+ \\
&\quad \sum_{r,p,t \in T^a, v \in \Theta_{VC}} CV_{r,p,t,v} \cdot \sum_{I,O} \mathbf{FOA}_{r,p,i,t,v,o} && \text{(annual variable cost on annual flows)} \\
&\quad \text{where } t \in T^a \\
&+ \\
C_{emissions} &= \sum_{r,p,e \in \Theta_{CE}} CE_{r,p,e} \cdot EAC_{r,e,i,t,v,o} \cdot \sum_{S,D,I,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} && \text{(annual emission cost on flow)} \\
&\quad \text{where } t \notin T^a \\
&+ \\
&\quad \sum_{r,p,e \in \Theta_{CE}} CE_{r,p,e} \cdot EAC_{r,e,i,t,v,o} \cdot \sum_{I,O} \mathbf{FOA}_{r,p,i,t,v,o} && \text{(annual emission cost on annual flows)} \\
&\quad \text{where } t \in T^a \\
&+ \\
&\quad \sum_{r,p,e \in \Theta_{CE}} \frac{CE_{r,p,e} \cdot EE_{r,e,t,v} \cdot \mathbf{NCAP}_{r,t,v=p}}{LEN_p} && \text{(annual embodied emission cost)} \\
&+ \\
&\quad \sum_{r,p,e \in \Theta_{CE,v}} CE_{r,p,e} \cdot EEOL_{r,e,t,v} \cdot \mathbf{ART}_{r,p,t,v} && \text{(annual retirement/end of life emission cost)}
\end{aligned} \tag{7.26}$$

Each of these costs are then discounted within each period and then to the base year:

$$\begin{aligned}
C_{fix,var,emiss} &= C_{fixed} + C_{variable} + C_{emissions} \\
&\quad \cdot \frac{P}{A} (i = GDR, N = LEN_p) && \text{(for each year in period } p \text{ discounted to NPV in } p) \\
&\quad \cdot \frac{P}{F} (i = GDR, N = p - P_0) && \text{(discounted from period } p \text{ to NPV in base year } P_0)
\end{aligned} \tag{7.27}$$

7.6.5 User-Specific Constraints

`temoa.components.limits.limit_emission_constraint` (*model: TemoaModel, r: Region, p: Period, e: Commodity, op: str*) $\rightarrow$ ExprLike

A modeler can track emissions through use of the `commodity_emissions` set and `emission_activity` parameter. The `EAC` parameter is analogous to the efficiency table, tying emissions to a unit of activity. The `limit_emission` constraint allows the modeler to assign an upper bound per period to each emission commodity. Note that this constraint sums emissions from technologies with output varying at the time slice and those with constant annual output in separate terms. It also includes embodied emissions from new capacity and end-of-life

emissions from retiring capacity.

$$\begin{aligned}
& \sum_{S,D,I,T,V,O|r,e,i,t,v,o \in EAC} (EAC_{r,e,i,t,v,o} \cdot \mathbf{FO}_{r,p,s,d,i,t,v,o}) \\
& + \sum_{I,T,V,O|r,e,i,t \in T^a, v,o \in EAC} (EAC_{r,e,i,t,v,o} \cdot \mathbf{FOA}_{r,p,i,t \in T^a, v,o}) \\
& + \sum_T \frac{EE_{r,e,t,v=p} \cdot \mathbf{NCAP}_{r,t,v=p}}{LEN_p} \\
& + \sum_{T,V} EEO L_{r,e,t,v} \cdot \mathbf{ART}_{r,p,t,v} \\
& \leq, \geq, \text{ or } = LE_{r,p,e}
\end{aligned} \tag{7.28}$$

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

`temoa.components.limits.limit_activity_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, op: str*) $\rightarrow$ ExprLike

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

$$\begin{aligned}
& \sum_{S,D,I,V,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \\
& \forall \{r, p, t \notin T^a\} \in \Theta_{\text{limit_activity}} \\
& + \sum_{I,V,O} \mathbf{FOA}_{r,p,i,t \in T^a, v,o} \\
& \forall \{r, p, t \in T^a\} \in \Theta_{\text{limit_activity}} \\
& \leq, \geq, \text{ or } = LA_{r,p,t}
\end{aligned} \tag{7.29}$$

`temoa.components.limits.limit_activity_share_constraint` (*model: TemoaModel, r: Region, p: Period, g1: Technology, g2: Technology, op: str*) $\rightarrow$ ExprLike

Limits the activity of a given technology or group as a fraction of another technology or group, summed over a period. This can be used to set, for example, a renewable portfolio scheme constraint.

$$\begin{aligned}
& \sum_{R_g \subseteq R, S, D, I, (T^{g1} \setminus T^a) \subseteq T, V, O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{R_g \subseteq R, I, (T^{g1} \cap T^a) \subseteq T, V, O} \mathbf{FOA}_{r,p,i,t,v,o} \\
& \leq, \geq, \text{ or } = \\
& LAS_{r,p,g1,g2} \cdot \sum_{R_g \subseteq R, S, D, I, (T^{g2} \setminus T^a) \subseteq T, V, O} \mathbf{FO}_{r,p,s,d,i,t,v,o} + \sum_{R_g \subseteq R, I, (T^{g2} \cap T^a) \subseteq T, V, O} \mathbf{FOA}_{r,p,i,t,v,o} \\
& \forall \{r, p, g1, g2\} \in \Theta_{\text{limit_activity_share}}
\end{aligned} \tag{7.30}$$

`temoa.components.limits.limit_capacity_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, op: str*) $\rightarrow$ ExprLike

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

$$\begin{aligned}
& \mathbf{CAPAVL}_{r,p,t} \leq, \geq, \text{ or } = LC_{r,p,t} \\
& \forall \{r, p, t\} \in \Theta_{\text{limit_capacity}}
\end{aligned} \tag{7.31}$$

`temoa.components.limits.limit_new_capacity_constraint` (*model: TemoaModel, r: Region, t: Technology, v: Vintage, op: str*) → ExprLike

The `limit_new_capacity` constraint sets a limit on the newly installed capacity of a given technology or group in a given vintage year.

$$\mathbf{NCAP}_{r,t,v} \leq, \geq, \text{ or } = \mathbf{LNC}_{r,t,v} \quad (7.32)$$

`temoa.components.limits.limit_capacity_share_constraint` (*model: TemoaModel, r: Region, p: Period, g1: Technology, g2: Technology, op: str*) → ExprLike

The `limit_capacity_share` constraint limits the available capacity of a given technology or technology group as a fraction of another technology or group.

`temoa.components.limits.limit_new_capacity_share_constraint` (*model: TemoaModel, r: Region, g1: Technology, g2: Technology, v: Vintage, op: str*) → ExprLike

The `limit_new_capacity_share` constraint limits the share of new capacity of a given technology or group as a fraction of another technology or group.

`temoa.components.limits.limit_resource_constraint` (*model: TemoaModel, r: Region, t: Technology, op: str*) → ExprLike

The `limit_resource` constraint sets a limit on the available resource of a given technology across all model time periods. Note that the indices for these constraints are region and tech.

$$\begin{aligned} & \sum_{P,S,D,I,V,O} \mathbf{FO}_{r,p,s,d,i,t \notin T^a,v,o} \\ & + \sum_{P,I,V,O} \mathbf{FOA}_{r,p,i,t \in T^a,v,o} \\ & \leq, \geq, \text{ or } = \mathbf{LS}_{r,t} \\ & \forall \{r, t\} \in \Theta_{\text{limit_resource}} \end{aligned} \quad (7.33)$$

`temoa.components.limits.limit_tech_input_split_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, i: Commodity, t: Technology, v: Vintage, op: str*) → ExprLike

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

`temoa.components.limits.limit_tech_output_split_constraint` (*model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage, o: Commodity, op: str*) → ExprLike

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 limit the 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. Under this constraint, only the technologies with variable output at the timeslice level (i.e., NOT in the `tech_annual` set) are considered.

The constraint is formulated as follows:

$$\sum_{I, t \notin T^a} \mathbf{FO}_{r,p,s,d,i,t,v,o} \leq, \geq, \text{ or } = TOS_{r,p,t,o} \cdot \sum_{I, O, t \notin T^a} \mathbf{FO}_{r,p,s,d,i,t,v,o} \quad (7.34)$$

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

`temoa.components.limits.limit_tech_input_split_annual_constraint` (*model: TemoaModel, r: Region, p: Period, i: Commodity, t: Technology, v: Vintage, op: str*) → ExprLike

Allows users to limit shares of commodity inputs to a process producing a single output. These shares can vary by model time period. See `limit_tech_output_split_annual_constraint` for an analogous explanation. Under this function, only the technologies with constant annual output (i.e., members of the `tech_annual` set) are considered.

`temoa.components.limits.limit_tech_output_split_annual_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, v: Vintage, o: Commodity, op: str*) → ExprLike

This constraint operates similarly to `limit_tech_output_split_constraint`. However, under this function, only the technologies with constant annual output (i.e., members of the `tech_annual` set) are considered.

$$\sum_{I, T^a} \mathbf{FOA}_{r,p,i,t \in T^a, v, o} \leq, \geq, \text{ or } = TOSA_{r,p,t,o} \cdot \sum_{I, O, T^a} \mathbf{FOA}_{r,p,i,t \in T^a, v, o} \quad (7.35)$$

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

`temoa.components.limits.limit_tech_input_split_average_constraint` (*model: TemoaModel, r: Region, p: Period, i: Commodity, t: Technology, v: Vintage, op: str*) → ExprLike

Allows users to limit shares of commodity inputs to a process producing a single output. Under this constraint, only the technologies with variable output at the timeslice level (i.e., NOT in the `tech_annual` set) are considered. This constraint differs from `limit_tech_input_split` as it specifies shares on an annual basis, so even though it applies to technologies with variable output at the timeslice level, the constraint only fixes the input shares over the course of a year.

`temoa.components.limits.limit_tech_output_split_average_constraint` (*model: TemoaModel, r: Region, p: Period, t: Technology, v: Vintage, o: Commodity, op: str*) → ExprLike

Allows users to limit shares of commodity outputs from a process. Under this constraint, only the technologies with variable output at the timeslice level (i.e., NOT in the `tech_annual` set) are considered. This constraint differs from `limit_tech_output_split` as it specifies shares on an annual basis, so even though it applies to technologies with variable output at the timeslice level, the constraint only fixes the output shares over the course of a year.

temoa.components.limits.limit_annual_capacity_factor_constraint (model: TemoaModel, r: Region, p: Period, t: Technology, v: Vintage, o: Commodity, op: str) → ExprLike

The limit_annual_capacity_factor sets an upper bound on the annual capacity factor from a specific process. The first portion of the constraint pertains to technologies with variable output at the time slice level, and the second portion pertains to technologies with constant annual output belonging to the tech_annual set.

$$\sum_{S,D,I} \mathbf{FO}_{r,p,s,d,i,t,v,o} \leq, \geq, \text{ or } = \quad \mathbf{LIMACF}_{r,t,v,o} \cdot \mathbf{CAP}_{r,p,t,v} \cdot \mathbf{C2A}_{r,t} \quad \forall \{r, p, t \notin T^a, v, o\} \in \Theta_{\text{limit_annual_capacity_factor}} \quad (7.36)$$

$$\sum_I \mathbf{FOA}_{r,p,i,t,v,o} \leq, \geq, \text{ or } = \quad \mathbf{LIMACF}_{r,t,v,o} \cdot \mathbf{CAP}_{r,p,t,v} \cdot \mathbf{C2A}_{r,t} \quad \forall \{r, p, t \in T^a, v, o\} \in \Theta_{\text{limit_annual_capacity_factor}}$$

temoa.components.limits.limit_seasonal_capacity_factor_constraint (model: TemoaModel, r: Region, p: Period, s: Season, t: Technology, op: str) → ExprLike

The limit_seasonal_capacity_factor sets an upper bound on the seasonal capacity factor from a specific technology. The first portion of the constraint pertains to technologies with variable output at the time slice level, and the second portion pertains to technologies with constant annual output belonging to the tech_annual set.

$$\sum_{D,I,V,O} \mathbf{FO}_{r,p,s,d,i,t,v,o} \leq, \geq, \text{ or } = \quad \mathbf{LIMSCF}_{r,s,t} \cdot \mathbf{CAPAVL}_{r,p,t} \cdot \mathbf{C2A}_{r,t} \cdot \mathbf{SFS}_s \quad \forall \{r, p, s, t \notin T^a\} \in \Theta_{\text{limit_seasonal_capacity_factor}} \quad (7.37)$$

$$\sum_{I,V,O} \mathbf{FOA}_{r,p,i,t,v,o} \cdot \mathbf{SFS}_s \leq, \geq, \text{ or } = \quad \mathbf{LIMSCF}_{r,s,t} \cdot \mathbf{CAPAVL}_{r,p,t} \cdot \mathbf{C2A}_{r,t} \cdot \mathbf{SFS}_s \quad \forall \{r, p, s, t \in T^a\} \in \Theta_{\text{limit_seasonal_capacity_factor}}$$

temoa.components.storage.limit_storage_fraction_constraint (model: TemoaModel, r: Region, p: Period, s: Season, d: TimeOfDay, t: Technology, v: Vintage, op: str) → ExprLike

This constraint is used if the users wishes to force a specific storage charge level for certain storage technologies and vintages at a certain time slice. User-specified storage charge levels that are sufficiently different from the optimal could impact the cost-effectiveness of storage.

s can be a season from the time_season set or a sequential season from the time_sequential_season set.

$$\frac{\mathbf{SL}_{r,p,s,d,t,v}}{\mathbf{SFS}_s \cdot \mathbf{DPP}} \leq, \geq, \text{ or } = \quad \mathbf{CAP}_{r,p,t,v} \cdot \mathbf{C2A}_{r,t} \cdot \frac{\mathbf{SD}_{r,t}}{24 \cdot \mathbf{DPP}} \cdot \mathbf{LSF}_{r,s,d,t} \quad \forall \{r, p, s, d, t, v\} \in \Theta_{\text{limit_storage_fraction}} \quad (7.38)$$

For seasonal storage technologies, the LHS becomes:

$$\mathbf{SSL}_{r,p,s_{seq},t,v} + \frac{\mathbf{SFS}_{s_{seq}}}{\mathbf{SFS}_{s^*}} \cdot \mathbf{SL}_{r,p,s^*,d,t,v}$$

`temoa.components.limits.limit_growth_capacity` (*model: TemoaModel, r: Region, p: Period, t: Technology, op: str, degrowth: bool = False*) → ExprLike

Constrain the change of capacity available between periods. Forces the model to ramp up and down the availability of new technologies more smoothly. Has constant (seed, $S_{r,t}$) and proportional (rate, $R_{r,t}$) terms. This can be defined for a technology group instead of one technology, in which case, capacity available is summed over all technologies in the group. In the first period, previous available capacity $\text{CAPAVL}_{r,p,t}$ is replaced by previous existing capacity, if any can be found.

Growth:

$$\text{CAPAVL}_{r,p,t} \leq, \geq, \text{ or } = S_{r,t} + (1 + R_{r,t}) \cdot \text{CAPAVL}_{r,p_{prev},t} \quad \forall \{r, p, t\} \in \Theta_{\text{limit_growth_capacity}} \quad (7.39)$$

Degrowth:

$$\text{CAPAVL}_{r,p_{prev},t} \leq, \geq, \text{ or } = S_{r,t} + (1 + R_{r,t}) \cdot \text{CAPAVL}_{r,p,t} \quad \forall \{r, p, t\} \in \Theta_{\text{limit_degrowth_capacity}}$$

`temoa.components.limits.limit_growth_new_capacity` (*model: TemoaModel, r: Region, p: Period, t: Technology, op: str, degrowth: bool = False*) → ExprLike

Constrain the change of new capacity deployed between periods. Forces the model to ramp up and down the deployment of new technologies more smoothly. Has constant (seed, $S_{r,t}$) and proportional (rate, $R_{r,t}$) terms. This can be defined for a technology group instead of one technology, in which case, new capacity is summed over all technologies in the group. In the first period, previous new capacity $\text{NCAP}_{r,t,v_{prev}}$ is replaced by previous existing capacity, if any can be found.

Growth:

$$\text{NCAP}_{r,t,v} \leq, \geq, \text{ or } = S_{r,t} + (1 + R_{r,t}) \cdot \text{NCAP}_{r,t,v_{prev}} \text{ where } v = p \quad \forall \{r, p, t\} \in \Theta_{\text{limit_growth_capacity}} \quad (7.40)$$

Degrowth:

$$\text{NCAP}_{r,t,v_{prev}} \leq, \geq, \text{ or } = S_{r,t} + (1 + R_{r,t}) \cdot \text{NCAP}_{r,t,v} \text{ where } v = p \quad \forall \{r, p, t\} \in \Theta_{\text{limit_degrowth_capacity}}$$

`temoa.components.limits.limit_growth_new_capacity_delta` (*model: TemoaModel, r: Region, p: Period, t: Technology, op: str, degrowth: bool = False*) → ExprLike

Constrain the acceleration of new capacity deployed between periods. Forces the model to ramp up and down the change in deployment of new technologies more smoothly. Has constant (seed, $S_{r,t}$) and proportional (rate, $R_{r,t}$) terms. It is recommended to leave the rate term empty as it would prevent the possibility of inflection in the rate of deployment. This constraint can be defined for a technology group instead of one technology, in which case, new capacity is summed over all technologies in the group. In the first period, previous new capacities are replaced by previous existing capacities, if any can be found.

Growth:

$$\text{NCAP}_{r,t,v_i} - \text{NCAP}_{r,t,v_{i-1}} \leq, \geq, \text{ or } = S_{r,t} + (1 + R_{r,t}) \cdot (\text{NCAP}_{r,t,v_{i-1}} - \text{NCAP}_{r,t,v_{i-2}}) \quad \text{where } v_i = p \quad \forall \{r, p, t\} \in \Theta_{\text{limit_growth_capacityDelta}}$$

Degrowth:

$$\text{NCAP}_{r,t,v_{i-1}} - \text{NCAP}_{r,t,v_{i-2}} \leq, \geq, \text{ or } = S_{r,t} + (1 + R_{r,t}) \cdot (\text{NCAP}_{r,t,v_i} - \text{NCAP}_{r,t,v_{i-1}}) \quad \text{where } v_i = p \quad \forall \{r, p, t\} \in \Theta_{\text{limit_degrowth_capacityDelta}} \quad (7.41)$$

7.7 General Caveats

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

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

COMPUTATIONAL IMPLEMENTATION

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

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

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

- Python is a scripting language; in general, scripts are an order of magnitude slower than an equivalent compiled program.
- Pyomo provides similar functionality, but because of its Python heritage, is **much** more verbose than GAMS, AMPL, or MathProg.

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

8.1 Anatomy of a Constraint

To help explain the Pyomo implementation, we discuss a single constraint in detail. Consider the Demand (7.6) constraint:

$$\sum_{I,T,V} \mathbf{FOA}_{r,p,i,t,v,dem} = DEM_{r,p,dem}$$

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

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

1. a constraint definition (in `temoa/core/model.py`),
2. the constraint implementation (in `temoa/components/`), and
3. (optional) sparse constraint index creation (in `temoa/components/`).

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/core/model.py`

```

1 self.demand_constraint = Constraint(
2     self.regions, self.time_optimize, self.commodity_demand,
3     rule=demand_constraint
4 )

```

In line 1, ‘`self.demand_constraint =`’ creates a place holder in the model object `self`, called ‘`demand_constraint`’. 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/components/`

```

...
1 def demand_constraint(model: TemoaModel, r: Region, p: Period, dem: Commodity) -> ExprLike:
  ↳ExprLike:
2     if (r, p, dem) in model.singleton_demands:
3         return Constraint.Skip
4
5     supply_annual = sum(
6         model.v_flow_out_annual[r, p, s_i, s_t, s_v, dem]
7         for s_t, s_v in model.commodity_up_stream_process[r, p, dem]
8         for s_i in model.process_inputs_by_output[r, p, s_t, s_v, dem]
9     )
10
11     demand_constraint_error_check(supply_annual, r, p, dem)
12
13     expr = supply_annual == value(model.demand[r, p, dem])
14     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.core.model`), followed by the argument list. Note that the argument list is strongly typed (each element has an explicit type like `Region`). Temoa enforces strong typing for code integrity using the `mypy` package. The argument list will always start with the model (Temoa convention shortens this to just `model`) 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 `model` will be an item from `region`, the second from `time_optimize`, and the third from `commodity_demand`. Though one could choose `a`, `b`, and `c` (or any naming scheme), we chose `r`, `p`, 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.core.model`, the constraint definition listed three index sets, so Pyomo will naively call this function for every possible combination of tuple $\{r, p, dem\}$. However, as there may be region-period-demand combinations for which the constraint is unnecessary, there is no need to create a constraint for such tuples.

Lines 5 through 9 sum the annual output flow of demand commodity `dem` across all technologies and vintages. The `supply_annual` is a local variable used in the expression (`expr`) shown below. Note that the sum is performed with sparse indices, which are returned from dictionaries created in `temoa/components/`.

Lines 5 through 9 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 model.tech_all:
    for S_v in model.vintage_all:
        for S_i in process_inputs_by_output( r, p, S_t, S_v, dem ):
            to_sum.append( model.v_flow_out_annual[r, p, S_i, S_t, S_v, dem] )
supply_annual = sum( to_sum )
```

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

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

```
supply_annual = 0
for S_t in model.tech_all:
    for S_v in model.vintage_all:
        for S_i in process_inputs_by_output( r, p, S_t, S_v, dem ):
            supply_annual += model.v_flow_out_annual[r, p, 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_annual` variable, Line 11 calls a function defined in `temoa/components/commodities.py` that checks to make sure there is a technology that can supply each demand commodity `dem` in each $\{r, p\}$.

If no process supplies the demand, then it quits computation immediately rather than completing a potentially lengthy model generation and waiting for the solver to recognize the infeasibility of the model. Further, the function lists potential ways for the modeler to correct the problem. This is one of the benefits of Temoa: we’ve incorporated error handling in several places to try and capture the most common user errors. This capability is subtle, but in practice extremely useful while building and debugging a model.

Line 12 creates the actual equality comparison. This line is superfluous, but we leave it in the code as a reminder that comparison operators (i.e. `==`, `<=`, `>=`) with a Pyomo object (like `supply_annual`) generate a Pyomo *expression object*, not a boolean True or False as one might expect [[return_expression](#)]. It is this expression object that must be returned to Pyomo, as on Line 13.

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

in `temoa/core/model.py` (declaration of indices, parameter, and constraint)

```
1 self.demand_constraint_rpc = Set(
2     within=self.regions * self.time_optimize * self.commodity_demand
```

```

3 )
4 self.demand = Param(self.demand_constraint_rpc)
5
6 ...
7
8 self.demand_constraint = Constraint(
9     self.demand_constraint_rpc, rule=commodities.demand_constraint
10 )

```

As discussed above, the demand_constraint is only valid for certain $\{r, p, dem\}$ tuples. Since not all combinations of region, period, and demand commodity may be valid, Temoa must ascertain the valid tuples. Thus, Line 1 tells Pyomo to instantiate demand_constraint_rpc as a Set of 3-length tuple indices within the dimensions of the regions, time_optimize, and commodity_demand sets, and populate it only with entries loaded from the database. Demand values are stored in the parameter self.demand declared in line 4. Data from the database is loaded into this index set and parameter via the HybridLoader class in temoa/data_io/hybrid_loader.py, declared in the component_manifest.py file.

in temoa/data_io/hybrid_loader.py (telling Temoa to load the indices)

```

1 def load_param_idx_sets(self, data: dict[str, object]) -> dict[str, object]:
2
3     model = TemoaModel()
4     param_idx_sets = {
5         ...
6         model.demand.name: model.demand_constraint_rpc.name, # param index sets
7         ...
8     }

```

in temoa/data_io/component_manifest.py (telling Temoa to load the parameter)

```

1 def build_manifest(model: TemoaModel) -> list[LoadItem]:
2
3 manifest = [
4     ...
5     LoadItem(
6         component=model.demand, # param name
7         table='demand', # database table name
8         columns=['region', 'period', 'commodity', 'demand'], # indices and value_
9         ↪column
10     ),
11     ...
12 ]

```

Alternatively, this index set could be populated using an index function which determines the valid indices from other model components. This second method is needed when the indices of a constraint do not exactly align with the indices of the database table. For example, the database table ramp_up_hourly is indexed by region, tech in the database but one of its inheriting constraints, ramp_up_day_constraint, is indexed by region, period, season, time_of_day, tech, vintage in the model. In this case, an index function is needed to determine the valid indices for the constraint.

in `temoa/core/model.py` (index set and constraint declaration)

```

1 self.ramp_up_day_constraint_rpsdtv = Set(
2     dimen=6, initialize=operations.ramp_up_day_constraint_indices
3 )
4 self.ramp_up_day_constraint = Constraint(
5     self.ramp_up_day_constraint_rpsdtv, rule=operations.ramp_up_day_constraint
6 )

```

in `temoa/components/operations.py` (return valid indices)

```

1 def ramp_up_day_constraint_indices(
2     model: TemoaModel,
3 ) -> set[tuple[Region, Period, Season, TimeOfDay, Technology, Vintage]]:
4     indices = {
5         (r, p, s, d, t, v)
6         for r, p, t in model.ramp_up_vintages
7         for v in model.ramp_up_vintages[r, p, t]
8         for s in model.time_season
9         for d in model.time_of_day
10    }
11
12    return indices

```

With the sparse set (`demand_constraint_rpc` or `ramp_up_day_constraint_rpsdtv`) created, we can now use it in place of the 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 `_rpc` 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 index function 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 `demand_constraint_rpc` implies that this set has a dimensionality of 3, and (following the [naming scheme](#)) the first index of each tuple will be an element of `region`, the second an element of `time_optimize`, and third a demand commodity.

8.2 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. demand_constraint{(r, p, dem) in sDemand_rpc} :
    sum{(r, p, Si, St, Sv, dem) in sFlowVar_rpitvo}
        v_flow_out_annual[r, p, Si, St, Sv, dem]
    =
        pDemand[r, p, dem];

```

While the syntax is not a direct translation, the indices of the constraint (`r`, `p`, and `dem`) are clear, and by inference, so are the indices of summation (`i`, `t`, `v`) and operand (`v_flow_out_annual`). This one-line definition creates an equality for each region, period, and demand, ensuring that total annual output meets each demand – almost exactly as we have formulated the demand constraint (7.6). In contrast, Temoa’s implementation in Pyomo takes many more 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 `demand_constraint_error_check` 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_annual == value(model.demand[r, p, 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 annual flows 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."

8.3 Execution Flow

The Temoa execution flow is designed to be modular and consistent across different modeling modes. It begins with the Command Line Interface (CLI), which parses user input and sets up the execution environment. The following sequence diagram illustrates the execution flow of a Temoa run, from the initial command-line invocation to the final processing of results.

The execution flow involves several key stages:

1. **CLI Entry:** The user interacts with `temoa/cli.py` to initiate a run or validation.
2. **Environment Setup:** The CLI creates a timestamped output directory, initializes logging, and uses `TemoaConfig` to parse the provided TOML configuration file. This includes checking for the availability of the specified optimization solver.
3. **Sequencing:** A `TemoaSequencer` is created. This object is the main coordinator, selecting the appropriate execution path based on the modeling mode (e.g., Perfect Foresight, Myopic, MGA).
4. **Data Loading:** The sequencer uses a `HybridLoader` to pull data from the SQLite database. This data is organized into a `Pyomo DataPortal`.

5. **Model Construction:** Using the `DataPortal`, Temoa constructs a `TemoaModel` instance. This stage builds all the mathematical sets, parameters, variables, and constraints.
6. **Solving:** The model instance is passed to `solve_instance()`, which invokes the chosen solver (like HiGHS, CBC, or Gurobi).
7. **Result Processing:** After the solver completes, `handle_results()` extracts the solution, checks for optimality, and persists the results back to the database. It also generates any requested auxiliary outputs like Excel files or network plots.

8.4 Project Structure

The Temoa model code is organized into clear, purpose-driven packages:

Core Packages:

- `temoa.core` - Public API for end users
 - `model.py` - Contains the `TemoaModel` class, the main entry point for building and solving energy system models. This class coordinates all model components.
 - `config.py` - Contains `TemoaConfig` and `TemoaMode` for model configuration and execution mode selection (perfect foresight, myopic, MGA, etc.).
- `temoa.cli` - Command-line interface
 - Provides the `temoa` command with subcommands for running models, validating configurations, migrating databases, and generating tutorial files.
- `temoa.components` - Model components and constraints
 - `costs.py` - Objective function implementation (total system cost minimization)
 - `flows.py` - Commodity flow balance constraints
 - `capacity.py` - Capacity and activity constraints
 - `emissions.py` - Emission accounting and constraints
 - `reserves.py` - Reserve margin requirements
 - `limits.py` - Various limit constraints (capacity, activity, emissions, etc.)
 - `storage.py` - Energy storage constraints
 - `operations.py` - Baseload and ramping constraints for generators
 - Additional constraint modules for specific features
- `temoa.data_io` - Data loading and validation
 - `hybrid_loader.py` - Main data loading engine using manifest-driven architecture
 - `component_manifest.py` - Declarative specification of all data components
 - `loader_manifest.py` - Data structure definitions for the loader
 - Database interface and validation logic
- `temoa.model_checking` - Model validation and integrity checking
 - Price checking for cost data consistency
 - Source tracing for commodity network validation
 - Network visualization tools

- `temoa.data_processing` - Output analysis and visualization
 - `db_to_excel.py` - Excel output generation ([!] untested in v4.0)
 - `make_graphviz.py` - Network diagram generation ([!] untested in v4.0)
 - Result processing utilities
- `temoa.extensions` - Optional extensions for advanced analysis
 - `modeling_to_generate_alternatives` - *Modeling to Generate Alternatives (MGA)* (MGA analysis for exploring near-optimal solutions)
 - `method_of_morris` - Sensitivity analysis ([!] untested in v4.0)
 - `monte_carlo` - *Monte Carlo Simulation* (Uncertainty quantification)
 - `myopic` - *Myopic Optimization* (Sequential decision making with limited foresight)
 - `single_vector_mga` - *Modeling to Generate Alternatives (MGA)* (Focused MGA on specific variables)
 - `stochastics` - *Stochastic Programming* (Stochastic programming capabilities)
- `temoa._internal` - Internal utilities (not part of public API)
 - `table_writer.py` - Database output formatting
 - `table_data_puller.py` - Result extraction utilities
 - Other internal helper modules

If you are working with a Temoa Git repository, these packages are in the `temoa/` subdirectory. For detailed architecture documentation, see the `README.md` file in the repository root.

8.5 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`
- * Cooprr 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_
->(existing_capacity)\n.')
+                                     'have any existing install base_
+>(existing_capacity).\n')
```

(continues on next page)

(continued from previous page)

```
SE.write( msg % str(l_process) )
continue
if 0 == M.existing_capacity[ l_process ]:
[ ... ]
```

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

8.5.1 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).

8.6 Ruff Formatting

The project has shifted to using Ruff (``ruff``) as a formatter / linter. Commits to the project should use Ruff to format any changed code. Ruff relies on settings in the `pyproject.toml` file. Contributors should be able to apply ruff to any changed files with the command `ruff format <file>`. If there is *specific* need to disable Ruff for a particular table or equation, contributors can sparingly turn off Ruff formatting for sections of code using comments. (See the Ruff documentation.)

8.7 Indentation: Tabs and Spaces

The project is standardized to using spaces for indentation in accordance with PEP-8 standards. Ruff will convert tabs to spaces.

8.8 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.

8.9 Maximum Line Length

(Similar to **PEP 8**) Limit all lines to a maximum of 100 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. While 80 characters remains an excellent metric of what constitutes a “long line” most modern wide-screen displays can comfortably show side-by-side difference files with 100 characters per side, and 100 characters better accommodates some long equations. A long line in this sense is one that is not as transparent as to its intent as it could be. **Ruff will enforce 100 character line length**, in accordance with the settings in the `pyproject.toml` file

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 )
```

8.10 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.

8.11 Encodings

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

8.12 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. `self.somename_constraint = Constraint( ..., rule=somename_constraint)`):

```
self.capacity_constraint = Constraint( self.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 `model` to represent the Pyomo model instance, `t` to represent *technology*, and `v` to represent *vintage*:

```
def capacity_constraint ( model: TemoaModel, t: Technology, v: Vintage ):
    ...
```

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.

8.13 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

>>> 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
```

8.14 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
```

8.15 Patches and Commits to the Repository

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

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

Fixed broken storage algorithm

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

License: MIT

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
```

(continues on next page)

(continued from previous page)

```
syntax[5].
```

```
License: MIT
```

- 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 MIT. A line like this at the end of your commit will suffice:

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

```
License: MIT
```

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 MIT license. 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 MIT 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 MIT.

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.

MODELING TO GENERATE ALTERNATIVES (MGA)

Temoa provides two extensions for Modeling to Generate Alternatives (MGA), an algorithm to explore the near-optimal solution space: hull expansion and single-vector MGA. Both are described in more detail below.

9.1 Hull Expansion

Hull expansion creates the convex hull containing all near-optimal solutions bound by the extreme points along a particular solution axis and then samples the continuum of solutions. The size of the convex hull is determined by the degree of cost relaxation (`cost_epsilon`). The method is described in [Pedersen et al. \(2021\)](#)

9.1.1 Configuration

To enable hull expansion, set the `scenario_mode` to "mga" in your configuration TOML file and provide an [MGA] section:

```
scenario_mode = "mga"

[MGA]
cost_epsilon = 0.05           # Relax cost by 5%
iteration_limit = 20           # Maximum number of MGA iterations
time_limit_hrs = 12           # Stop after 12 hours
axis = "TECH_CATEGORY_ACTIVITY"
weighting = "HULL_EXPANSION"
```

Options

- **cost_epsilon**: The fraction by which the optimal cost is allowed to increase (e.g., 0.05 for 5%).
- **iteration_limit**: The maximum number of alternative solutions to generate.
- **time_limit_hrs**: The maximum wall-clock time for the entire MGA run.
- **axis**: The dimension along which to optimize. Supported values:
 - TECH_CAPACITY
 - TECH_CATEGORY_CAPACITY
 - TECH_CATEGORY_ACTIVITY (Default)
 - EMISSION_ACTIVITY
- **weighting**: The algorithm used to select the next optimization vector. Currently, only `HULL_EXPANSION` is supported.

9.2 Single-Vector MGA (SVMGA)

Single-Vector MGA is a simplified, two-stage process. First, it solves the base model to find the minimum cost. Second, it adds a cost relaxation constraint and creates a new objective function that minimizes user-specified quantities, such as technology-specific installed capacity or total carbon dioxide emissions.

9.2.1 Configuration

To enable SVMGA, set the `scenario_mode` to "svmga" and provide an `[SVMGA]` section:

```
scenario_mode = "svmga"

[SVMGA]
cost_epsilon = 0.05
capacity_labels = ["solar_pv", "wind_onshore"]
# emissions_labels = ["CO2"]
# activity_labels = ["coal_power"]
```

Options

- **cost_epsilon**: Same as in hull expansion.
- **capacity_labels**: A list of technology names whose total capacity should be maximized in the second stage. Matching is **exact and case-sensitive** against the identifiers in the `tech_all` set. Example: ["solar_pv", "wind_onshore"].
- **emissions_labels**: A list of emission commodities whose total emissions should be minimized. Matching is **exact and case-sensitive** against identifiers in the `commodity_emissions` set. Example: ["CO2"].
- **activity_labels**: A list of technology names whose total activity (energy flow out) should be maximized. Matching is **exact and case-sensitive**. Example: ["coal_power"].

Note: SVMGA will construct an unweighted sum of all variables matching these labels as the new objective function. Be careful not to mix different units. In addition, note that the MGA objective function is set to minimize regardless of the label choice.

9.3 Parallel Execution and Solver Options

Standard MGA supports parallel execution of iterative solves to maximize performance. **Note: SVMGA executes sequentially and does not utilize parallel workers.**

The number of worker processes and solver-specific settings are defined in a `MGA_solver_options.toml` file. By default, Temoa looks for this file in the same directory as your main configuration file.

```
# Global setting at the top level of the file
num_workers = 4

[gurobi]
Method = 2
Threads = 4 # Threads per solver instance
BarConvTol = 0.01
```

 Tip

When choosing `num_workers`, a good rule of thumb is to set it to the number of available CPU cores minus one. This leaves room for the main orchestration process and ensures that the system remains responsive. Also, be mindful of the `Threads` setting within solver blocks, as the total thread count will be `num_workers * Threads`.

9.4 Outputs

MGA results are stored in the same output database specified in your configuration. Each iteration is saved as a unique scenario to allow for easy comparison and analysis.

9.4.1 Scenario Naming Convention

Each run is saved under a unique scenario name in the output tables, following the format: `<base_scenario>-<iteration_index>`.

- **Iteration 0:** The original baseline solve (optimal solution).
- **Iterations 1-N:** The alternative solutions generated by the MGA algorithm.

For example, if your base scenario is `utopia_mga`, the results for the base case will be found under scenario `utopia_mga-0`, and the first alternative will be under `utopia_mga-1`.

9.4.2 Key Database Tables

The results are spread across several tables, consistent with standard Temoa runs:

- **output_objective:** Stores the total system cost and MGA optimization objective for each iteration.
- **output_net_capacity:** Stores the installed capacity for each technology, period, and iteration.
- **output_flow_out / output_flow_out_summary:** Stores energy flows between technologies.
- **output_emission:** Stores emission results per commodity and technology.
- **output_cost:** Stores detailed cost breakdowns (investment, fixed, variable).

9.4.3 Comparing Iterations

You can use SQL queries to compare results across different MGA iterations.

Comparing Total System Cost:

```
SELECT scenario, total_system_cost
FROM output_objective
WHERE scenario LIKE 'utopia_mga-%'
ORDER BY scenario;
```

Comparing Capacity for a Specific Technology:

```
SELECT scenario, tech, period, capacity, units
FROM output_net_capacity
WHERE scenario LIKE 'utopia_mga-%'
AND tech = 'solar_pv'
ORDER BY scenario, period;
```

Analyzing Diversity (SQL Join Example):

```
SELECT a.scenario, a.tech, a.capacity as cap_a, b.capacity as cap_b, (a.capacity -  
↪b.capacity) as diff  
FROM output_net_capacity a  
JOIN output_net_capacity b ON a.tech = b.tech AND a.period = b.period  
WHERE a.scenario = 'utopia_mga-1'  
      AND b.scenario = 'utopia_mga-0'  
      AND a.tech = 'solar_pv';
```

MONTE CARLO SIMULATION

Temoa provides a Monte Carlo simulation framework that allows users to perform probabilistic analysis by executing multiple model runs with varying input parameters. This feature is useful for characterizing the distribution of outcomes under uncertainty or performing sensitivity analysis.

10.1 Overall Framework

The Monte Carlo extension in Temoa is designed to:

- Execute an arbitrary number of structured runs.
- Provide a clean interface to specify parameter deviations for each run.
- Support multiple deviations (tweaks) per individual run.
- Record all parameter adjustments in the output database for verification.
- Utilize parallel processing to speed up the execution of multiple runs.

10.2 Configuration

To enable Monte Carlo mode, set the `scenario_mode` to "monte_carlo" in your configuration TOML file and provide the path to the run settings file under a `[monte_carlo]` section:

```
scenario_mode = "monte_carlo"

[monte_carlo]
run_settings = "path/to/mc_settings.csv"
solver_options = "mc_solver_options.toml" # Optional
```

10.2.1 Settings

- **run_settings**: Path to the CSV file containing the parameter deviations.
- **solver_options** (Optional): Path to a TOML file containing worker counts and solver-specific options. If not provided, Temoa uses the default options file in `temoa/extensions/monte_carlo/MC_solver_options.toml`.

10.3 Input File Format

The Monte Carlo settings are defined in a CSV file. The file must contain a header and follow this structure:

```
run,param,index,mod,value,notes
1,cost_invest,utopia|TXD|2010,a,-44.0,reduce invest cost to 1000
2,demand,utopia|2010|RH,r,0.1,increase demand by 10%
2,cost_fixed,utopia|TXD|2010|2010,s,15.0,set fixed cost to 15.0
```

10.3.1 Columns

- **run:** An integer representing the run index. Multiple lines with the same run index will be applied to the same model instance.
- **param:** The name of the Temoa parameter to adjust (e.g., `cost_invest`, `demand`).
- **index:** The index of the parameter, using the pipe (`|`) character as a separator.
- **mod:** The adjustment type (see below).
- **value:** The numeric value used for the adjustment.
- **notes:** A free-text field for user notes.

10.3.2 Adjustment Types

The `mod` column supports three types of adjustments:

- **a (Absolute Change):** Adds the `value` to the baseline parameter value. Formula: $new_value = old_value + value$
- **r (Relative Change):** Adjusts the baseline value by a percentage. Formula: $new_value = old_value \times (1 + value)$
- **s (Substitution):** Replaces the baseline value with the provided `value`. Formula: $new_value = value$

10.3.3 Advanced Indexing

- **Wildcards:** You can use an asterisk (`*`) as a wildcard in any position of the `index` to apply the adjustment to all matching elements. Example: `utopia|*|2010` would match all regions for the given tech/commodity in 2010.
- **Multi-tokens:** You can specify multiple identifiers for a single index position by separating them with a forward slash (`/`). This will create a Cartesian product of all specified combinations. Example: `utopia/usa|TXD|2010` would apply the tweak to both `utopia|TXD|2010` and `usa|TXD|2010`.

10.4 Parallel Execution

Monte Carlo runs are executed in parallel to maximize performance. The number of worker processes and solver-specific options for these workers are controlled by a configuration file.

By default, Temoa uses the `MC_solver_options.toml` file located in the `temoa/extensions/monte_carlo/` directory. However, you can provide your own file by specifying the `solver_options` path in your configuration TOML.

Tip

The default configuration uses 11 worker processes. If you provide a custom `solver_options` file, you can adjust the `num_workers` setting and add solver-specific parameters (e.g., threads, tolerances) for each solver.

10.5 Outputs

The results of each Monte Carlo run are stored in the output database specified in your configuration.

10.5.1 Model Scenarios

Each run is saved under a unique scenario name in the output tables, following the format: `<base_scenario>-<run_index>`. For example, if your base scenario is `utopia_mc`, the results for the first run will be labeled `utopia_mc-1`.

10.5.2 Tweak Log Table

The specific adjustments made for each run are recorded in the `output_mc_delta` table. This table contains the following columns:

- **scenario:** The unique scenario name for the run.
- **run:** The run index.
- **param:** The parameter that was adjusted.
- **param_index:** The specific index that was tweaked.
- **old_val:** The original value from the baseline data.
- **new_val:** The new value after the adjustment was applied.

MYOPIC OPTIMIZATION

Myopic (or sequential) optimization in Temoa provides a way to solve the model in a series of smaller, sequential time windows. This is often used to simulate decision-making with limited foresight, where the model only “sees” a limited number of future time periods at any given step.

11.1 Overall Framework

The Myopic extension in Temoa is designed to:

- Solve the model for a sequence of overlapping or non-overlapping time windows.
- Limit the foresight of the model to a user-specified number of future periods.
- Fix the capacity investment decisions from previous windows as “initial capacity” for subsequent windows.
- Automatically manage the `myopic_efficiency` table to filter active technologies based on their remaining life-time and previous investment decisions.

11.2 Configuration

To enable Myopic mode, set the `scenario_mode` to “myopic” in your configuration TOML file and provide the required settings under a `[myopic]` section:

```
scenario_mode = "myopic"

[myopic]
view_depth = 2
step_size = 1
evolving = false
evolution_script = "temoa/extensions/myopic/evolution_updater.py"
```

11.2.1 Settings

- **view_depth:** The number of future time periods visible to the model in each iteration. For example, a `view_depth` of 2 means the model will optimize over two periods at a time.
- **step_size:** The number of periods to advance the “base year” in each subsequent iteration. A `step_size` of 1 means the base year will move forward by one time period at a time.
- **evolving:** A boolean flag (`true` or `false`) that selects between evolving and non-evolving myopic mode. Defaults to `false`. See *Evolving vs. Non-Evolving Mode* below.
- **evolution_script:** Path to a Python script containing an `iterate()` function that is called between iterations in evolving mode. Ignored when `evolving = false`.

Important

Myopic mode requires that the `input_database` and `output_database` in your configuration point to the same file. This is because the sequencer needs to write intermediate results (like the `myopic_efficiency` table and capacity decisions) back to the database to inform subsequent windows.

11.3 Evolving vs. Non-Evolving Mode

Temoa supports two myopic sub-modes that control how iterations are generated and whether the database can be modified between solves.

11.3.1 Non-Evolving Mode (default)

When `evolving = false`, the sequencer eliminates **redundant iterations** near the end of the planning horizon. If multiple consecutive iterations would all see the same final period, only the first such iteration is kept. This avoids re-solving windows that cannot produce new information.

For example, consider a model with future periods `[2025, 2030, 2035, 2040, 2045]`, `view_depth = 3`, and `step_size = 1`:

- Without pruning, the base years would be `[2025, 2030, 2035, 2040]`.
- The iterations starting at 2035 and 2040 both see period 2045 as the last demand year, so only the first is kept.
- **Result:** 3 iterations instead of 4.

Non-evolving mode is appropriate when the model structure (technologies, constraints, costs, etc.) does not change between iterations and you want to minimize computation.

11.3.2 Evolving Mode

When `evolving = true`, the sequencer keeps **all** calculated iterations, stepping forward by exactly `step_size` until the planning horizon is exhausted. Between each iteration, the `evolution_script` is called, giving the modeler an opportunity to modify the database before the next window is solved.

Using the same example (periods `[2025, 2030, 2035, 2040, 2045]`, `view_depth = 3`, `step_size = 1`):

- Base years are `[2025, 2030, 2035, 2040]`.
- **All 4 iterations** are kept, because the evolution script may change data between them.

This mode is designed for scenarios where model parameters—such as technology costs, demand levels, or policy constraints—need to evolve over time based on previous results or external logic.

Note

In evolving mode, the view depth may shorten near the end of the planning horizon so that the window does not extend beyond the available periods. All iterations are still performed.

11.3.3 Writing an Evolution Script

The evolution script must define an `iterate()` function with the following signature:

```
import sqlite3
from temoa.extensions.myopic.myopic_index import MyopicIndex

def iterate(
    idx: MyopicIndex | None = None,
    prev_base_year: int | None = None,
    last_instance_status: str | None = None,
    db_con: sqlite3.Connection | None = None,
) -> None:
    """Called between myopic iterations (evolving mode only)."""
    # Use db_con to read/write the database as needed.
    pass
```

Parameters:

- **idx:** The `MyopicIndex` for the current iteration, containing the base year, step year, last demand year, and last year.
- **prev_base_year:** The base year of the previous iteration.
- **last_instance_status:** 'optimal' if the previous solve succeeded, or 'roll_back' if it failed and triggered a rollback.
- **db_con:** An open SQLite connection to the myopic database, which can be used to read results from the prior iteration and modify data for the next one.

The function is called *after* results from the current iteration have been written to the database and *before* the myopic efficiency table is updated for the next iteration. A template is provided at `temoa/extensions/myopic/evolution_updater.py`.

11.4 How it Works

When running in Myopic mode, Temoa performs the following steps:

1. **Initialization:** It identifies all future time periods and sets up a queue of “optimization windows” based on the `view_depth` and `step_size`.
2. **Initial Setup:** It creates the `myopic_efficiency` table and pre-loads it with any existing capacity from the input database.
3. **Sequential Solve:** For each window in the queue:
 - a. **Update Efficiency Table:** The `myopic_efficiency` table is updated to include only technologies that are still within their lifetime or were built in previous windows. Any “planned” technologies not built in the previous window’s look-ahead are removed to prevent “ghost” capacity.
 - b. **Data Loading:** Data is loaded specifically for the current window’s time range, using the `myopic_efficiency` table as a filter.
 - c. **Model Solve:** A standard Temoa model instance is built and solved for the current window.
 - d. **Result Persistence:** Investment decisions (built and retired capacity) are saved back to the database. These decisions are treated as fixed constraints in all future windows.
4. **Final Reporting:** Once all windows are solved, a total system cost is calculated as the sum of discounted costs across all periods.

11.5 Myopic Efficiency Table

The `myopic_efficiency` table is a key internal component of the myopic sequencer. It acts as a dynamic version of the standard `efficiency` table, filtered for each optimization window.

- It is “actively maintained” during the run.

- Items not built in a previous time window are removed.
- Items that reach the end of their technical lifetime are retired and removed from the table.
- The table includes a computed `lifetime` field used internally to screen active technologies.

11.6 Backtracking and Roll-back

If the solver fails to find an optimal solution for a given window (e.g., due to infeasibility caused by previous decisions), the Myopic sequencer has a built-in roll-back mechanism. It will attempt to back up to the previous window and re-solve with an expanded `view_depth` to find a feasible path forward. If it cannot back up further, the run will abort with an error.

11.7 Notes and Caveats

11.7.1 Performance and Discounting

- **Reduced Compute:** Myopic mode dramatically reduces compute requirements compared to perfect foresight by using a “divide and conquer” approach, solving several smaller problems instead of one large global optimization.
- **Discounting:** All costs are discounted to the first optimization period in the database (period `p0` of the first myopic window).

11.7.2 Modeling Considerations

- **Investment Amortization:** Investment costs in Temoa are amortized over the useful life of a built technology. This harmonizes how the model perceives costs and benefits within the `view_depth`: since the model only sees the benefits of a technology for the portion of its life that falls within the current window, the costs are similarly distributed. Consequently, discounted investment costs reported in summary logs may be significantly higher than those reported in the `output_cost` table.
- **Limited Foresight:** Myopic mode cannot see beyond the specified `view_depth`, including constraints. If a constraint only becomes active later in the planning horizon, the model may make sub-optimal decisions in early time windows. This can lead to unexpected rollbacks when those constraints finally enter the visibility window, which can complicate the interpretation of the results. Modeler caution is advised when setting the `view_depth` relative to known future constraints.

STOCHASTIC PROGRAMMING

The Stochastics extension in Temoa v4 provides support for stochastic programming using the [mpi-sppy](#) library. This allows for decision-making under uncertainty by considering multiple scenarios simultaneously and finding an optimal “first-stage” decision that minimizes the expected cost over all scenarios.

Stochastic programming is particularly useful for modeling uncertainties in future costs, demands, or resource availability.

12.1 Dependencies

The stochastics extension requires the `mpi-sppy` package. You can install it using `uv`:

```
uv add mpi-sppy
```

Or using `pip`:

```
pip install mpi-sppy
```

12.2 Configuration

To run Temoa in stochastic mode, you need to modify your main configuration TOML file and provide an additional stochastic configuration file.

12.2.1 Main Configuration TOML

Set the `scenario_mode` to `"stochastic"` and add a `[stochastic]` section:

```
scenario_mode = "stochastic"

# ... other standard options ...

[stochastic]
# Path to the stochastic configuration file, relative to this file
stochastic_config = "stochastic_config.toml"
```

12.2.2 Stochastic Configuration TOML

The stochastic configuration file defines the scenarios, their probabilities, and the data perturbations associated with each scenario.

```
# Define the scenarios
[scenarios]
# Each key is a scenario name, and the value is its probability
# Probabilities must sum to 1.0
low_cost = 0.5
high_cost = 0.5

# Define perturbations for a specific scenario
[[perturbations]]
scenario = "low_cost"
table = "cost_variable"
# Filter specifies which rows in the table to perturb
filter = { tech = "IMPHCO1" }
# Action can be "multiply", "add", or "set" (defaults to "set")
action = "multiply"
value = 0.5

[[perturbations]]
scenario = "high_cost"
table = "cost_variable"
filter = { tech = "IMPHCO1" }
action = "multiply"
value = 1.5
```

Perturbation Options

Currently, the following fields are required for each perturbation:

- **scenario:** The name of the scenario to which this perturbation applies.
- **table:** The Temoa parameter (database table) to perturb (e.g., `cost_variable`, `demand`, `capacity_factor_process`).
- **filter:** A dictionary of column-value pairs used to identify specific rows. Since the extension uses the dynamic manifest from `HybridLoader`, any column belonging to the table's index can be used for filtering.
- **action:** The operation to perform. Supported values:
 - `multiply`: Multiply the base value by `value`.
 - `add`: Add `value` to the base value.
 - `set`: Replace the base value with `value`.
- **value:** The numeric value used in the perturbation action.

12.3 How it Works

When running in stochastic mode, Temoa:

1. Loads the base data from the input database.
2. Identifies the “first-stage” variables. In the current implementation, all decisions in the first time period are considered first-stage.
3. Orchestrates multiple scenario runs using the `mpi-sppy` Extensive Form (EF) solver.

4. For each scenario, the `scenario_creator` applies the specified perturbations to the base data and builds a Pyomo model instance.
5. The EF solver binds the first-stage variables across all scenarios (non-anticipativity constraints) and optimizes the total expected cost.
6. The terminal output reports the Stochastic Expected Value.

12.4 Limitations

- **Two-Stage Only:** While `mpi-sppy` supports multi-stage stochastic programming, the current Temoa integration is tailored for two-stage problems where the first time period constitutes the first stage.
- **Result Persistence:** Currently, only the expected objective value and summary logs are produced. Detailed per-scenario result persistence to the database is under development.

UNITS CHECKING

The Temoa v4.0 database schema includes comprehensive units checking capabilities to ensure consistency and accuracy throughout the model. Unit checking validates that all units are properly formatted, dimensionally consistent, and align correctly across related tables.

Important

The units expressed and checked via `pint` do not “follow the values” through the mathematics of the model. Unit checking is a **pre-processing validation layer** used to support documentation and catch input errors. **The units are not used in the model calculations themselves.**

13.1 Unit Propagation to Outputs

Starting with v4.0, units defined in input tables are **automatically propagated** to output tables when model results are written. This provides traceability of units from inputs through results.

How It Works:

- `output_flow_out / output_flow_in`: Units from `commodity.units` for the output/input commodity
- `output_built_capacity / output_net_capacity / output_retired_capacity`: Units from `existing_capacity.units`
- `output_emission`: Units from `commodity.units` for the emission commodity
- `output_cost`: Common currency unit extracted from cost input tables
- `output_storage_level`: Units from the stored commodity

Backward Compatibility:

- Databases without unit data in input tables will write `NULL` units to outputs
- Databases with older schemas (no `units` column on output tables) continue to work normally
- Unit propagation is automatic and requires no configuration

13.2 Overview

The unit checking system uses the Python package `pint` to perform unit validation and dimensional analysis. This leverages `pint`'s built-in unit registry to validate units with varying prefixes (e.g., PJ, TJ, GJ) and enables future extensions.

The basis for most unit comparisons comes from:

- **commodity table**: Defines native units for each commodity (energy network nodes)

- **efficiency table:** Infers technology units via input/output ratios
- **capacity_to_activity table:** Provides conversion factors for capacity-based measures

13.3 Enabling Unit Checking

13.3.1 Via Configuration File

Add to your `config.toml`:

```
check_units = true
```

13.3.2 Via CLI

Standalone unit checking for any database:

```
# Basic usage
temoa check-units path/to/database.sqlite

# Custom output directory
temoa check-units database.sqlite --output ./reports

# Silent mode (for scripting)
temoa check-units database.sqlite --silent
```

13.4 How It Works

The unit checker performs five sequential tests:

1. **Database Version Check:** Ensures database is v4.0+ (checks `metadata` table)
2. **Units Entry Validation:** Checks for illegal characters, proper formatting, registry membership
3. **Technology I/O Alignment:** Validates `efficiency` table units match commodities
4. **Related Tables:** Checks tables referencing technologies for unit consistency
5. **Cost Tables:** Validates cost units and dimensional alignment

13.5 Expressing Units

13.5.1 Format Requirements

Warning

CRITICAL: The unit checker uses regex parsing with strict format requirements!

Units in **efficiency** and **cost** tables **MUST** use ratio format:

```
Numerator / (Denominator)

[V] CORRECT:    PJ / (PJ)
```

(continues on next page)

(continued from previous page)

```
[V] CORRECT:  Mdollar / (PJ^2 / GW)
[X] WRONG:    PJ/PJ              (no parentheses)
[X] WRONG:    Mdollar * GW / (PJ^2)  (denominator incomplete)
```

The denominator **MUST be fully enclosed in parentheses**. The regex only captures content within () after the /.

Other tables should use plain entries:

```
[V] PJ
[V] petajoules
[V] GW
[V] Mt / (GW)    (if ratio needed)
```

13.5.2 Custom Units Registry

Temoa extends pint's default registry with domain-specific units:

- dollar (or USD)
- euro (or EUR)
- passenger
- seat (for passenger-miles, seat-miles)
- ethos (dimensionless source commodity)

13.6 Common Footguns and Pitfalls

13.6.1 1. Missing Parentheses in Ratios

Problem: Forgetting to parenthesize the denominator

```
[X] cost_invest units: "Mdollar / PJ^2 / GW"

Error: RATIO_ELEMENT regex doesn't match, only "PJ^2" captured
```

Solution: Always use parentheses

```
[V] cost_invest units: "Mdollar / (PJ^2 / GW) "
```

13.6.2 2. Capacity vs Energy Units

Problem: Using energy units (GW*year, kWh) in capacity tables

```
[X] existing_capacity units: "GW * year"

Error: Energy units (not capacity) in capacity table
```

Solution: Use power units (negative time dimension)

```
[V] existing_capacity units: "GW"      ([time]^-3 = power)
[X] existing_capacity units: "GWh"     ([time]^+2 = energy)
```

Physics: - Capacity = Power (W, kW, MW, GW) -> [time]⁻³ - Energy = Power × Time (Wh, kWh) -> [time]⁺²

13.6.3 3. Cost Table Units and C2A

Problem: Not accounting for capacity_to_activity conversion

For capacity-based costs, the expected denominator is:

$$\text{expected_measure} = \text{output_units} \times \text{C2A} \times \text{year}$$

Example: - Output commodity: ELC = PJ - C2A: PJ / (GW * year) - Expected: PJ * [PJ / (GW*year)] * year = PJ^2 / GW

```
[V] cost_invest: "Mdollar / (PJ^2 / GW) "
[V] cost_fixed:  "Mdollar / (PJ^2 / GW / year) "    (adds /year for period-based)
```

13.6.4 4. period_based Flag

Problem: Misunderstanding period_based vs capacity_based

These flags are **orthogonal**:

- capacity_based: Multiply expected units by C2A
- period_based: Divide expected units by year

cost_fixed is **BOTH** capacity_based AND period_based!

Table	capacity_based	period_based	Units Example
cost_invest	True	False	Mdollar / (PJ^2 / GW)
cost_fixed	True	True	Mdollar / (PJ^2 / GW / year)
cost_variable	False	False	Mdollar / (PJ)
cost_emission	False	False	Mdollar / (Mt)

13.6.5 5. Schema Variations: tech vs tech_or_group

Problem: v4.0 limit tables use tech_or_group not tech

The unit checker automatically detects which column exists, but be aware:

```
-- v3.1 schema
SELECT tech, units FROM limit_capacity...

-- v4.0 schema
SELECT tech_or_group, units FROM limit_capacity...
```

Tables affected: limit_activity, limit_capacity, limit_new_capacity

13.6.6 6. Technology Output Uniformity

Problem: Same technology with multiple output commodities using different units

Warning

All instances of a technology **MUST** produce output in the **same units**, even if output commodities differ!

Why: Constraints span regions and output commodities, requiring a single unit of measure.

```
[X] WRONG:
tech E01, output ELC: PJ
tech E01, output HEAT: GJ      (different units!)

[V] CORRECT:
tech E01, output ELC: PJ
tech E01, output HEAT: PJ      (same units)
```

13.7 Testing and Troubleshooting

13.7.1 Testing Units Manually

Test unit validity outside the model:

```
from temoa.model_checking.unit_checking import ureg

# Check if units exist in registry
print('PJ' in ureg)      # True
print('catfood' in ureg) # False

# Parse and check units
u = ureg('Mdollar / (PJ^2 / GW)')
print(u.dimensionality)  # {[currency]: 1, [time]: 1, ...}

# Check currency dimension (for cost tables)
print('[currency]' in u.dimensionality) # True
```

13.7.2 Common Error Messages

```
"Units lack currency dimension"
-> Cost table units don't contain currency (dollar/euro)

"Energy units (not capacity) in capacity table"
-> Using energy units (kWh, GWh, etc.) instead of capacity units (GW, MW)

"Non-matching measure unit"
-> Units don't match expected format after accounting for C2A/period

"failed to process query: no such column: tech"
-> Using old SQL queries on v4.0 schema (should auto-fix now)
```

13.7.3 Reading Unit Check Reports

Reports are saved to `output_path/unit_check_reports/units_check_TIMESTAMP.txt`:

```
===== Units Check 1 (DB Version): Started =====
Units Check 1 (DB Version): Passed

===== Units Check 2 (Units Entries in Tables): Started =====
existing_capacity: Energy units (not capacity) in capacity table: GW*year at rows: 1, 2, 3
```

(continues on next page)

(continued from previous page)

```

===== Units Check 3 (Tech I/O via Efficiency Table): Started =====
Efficiency units conflict with associated commodity for Technology E01 near row 1

===== Units Check 4 (Related Tables): Started =====
limit_capacity: Non-standard units for tech E01 (expected GW) got: MW at rows: 5, 6

===== Units Check 5 (Cost Tables): Started =====
cost_invest: Non-matching measure unit for tech/comm: E01
  Table entry: Mdollar / (PJ)
  Expected: petajoule ** 2 / gigawatt
  Found: petajoule at rows: 1, 2, 3

```

Each check section shows: - Which table/tech has issues - What was expected vs what was found - Row numbers for easy correction

13.8 Tables Checked

13.8.1 v4.0 Schema Coverage

Table	Has Units	Check 2	Check 3	Check 4	Check 5
capacity_to_activity	[V]	[V]		(used)	(used)
commodity	[V]	[V]	[V]		
construction_input	[V]	[V]			
cost_emission	[V]	[V]			[V]
cost_fixed	[V]	[V]			[V]
cost_invest	[V]	[V]			[V]
cost_variable	[V]	[V]			[V]
demand	[V]	[V]		[V]	
efficiency	[V]	[V]	[V]		
emission_activity	[V]	[V]			
emission_embodied	[V]	[V]			
emission_end_of_life	[V]	[V]			
end_of_life_output	[V]	[V]			
existing_capacity	[V]	[V]		[V]	
lifetime_process	[V]	[V]			
lifetime_tech	[V]	[V]			
loan_lifetime_process	[V]	[V]			
limit_activity	[V]	[V]		[V]	
limit_capacity	[V]	[V]		[V]	
limit_emission	[V]	[V]			
limit_new_capacity	[V]	[V]		[V]	
limit_resource	[V]	[V]			

Check Legend: - Check 2: Standard validation (format, characters, registry) - Check 3: Technology I/O alignment via Efficiency table - Check 4: Related tables consistency - Check 5: Cost tables validation

13.9 Best Practices

1. **Start with commodities:** Define commodity units first, then build tech efficiency ratios
2. **Use standard prefixes:** Stick to k, M, G, T prefixes (kilo, mega, giga, tera)
3. **Be consistent:** Use the same unit style across your database (e.g., always “PJ” not mix of “PJ”/“petajoule”)
4. **Test early:** Run unit checker on partial databases during development
5. **Document assumptions:** Use notes fields to explain unusual unit choices
6. **Reference implementation:** See `temoa/tutorial_assets/utopia.sql` for a fully compliant example

13.10 Quick Reference

13.10.1 Common Unit Patterns

```
# Energy commodities
commodity units: PJ, TJ, GJ, MWh

# Capacity
existing_capacity units: GW, MW, kW

# Efficiency (energy tech)
efficiency units: PJ / (PJ)          (dimensionless)
efficiency units: PJ / (Mt)          (energy from mass)

# C2A conversion
capacity_to_activity units: PJ / (GW * year)

# Costs (capacity-based tech with PJ output, PJ/(GW*year) C2A)
cost_invest units: Mdollar / (PJ^2 / GW)
cost_fixed units: Mdollar / (PJ^2 / GW / year)
cost_variable units: Mdollar / (PJ)

# Emissions
emission_activity units: Mt / (PJ)
emission_embodied units: Mt / (GW)
```

13.10.2 Dimension Reference

Pint tracks seven base dimensions:

- [length]: meter, km, mile
- [mass]: kg, tonne, Mt
- [time]: second, year
- [current]: ampere
- [temperature]: kelvin
- [substance]: mole
- [luminosity]: candela
- [currency]: dollar, euro (Temoa extension)

Derived dimensions:

- Energy: $[\text{length}]^2 * [\text{mass}] / [\text{time}]^2$ (joule, kWh)
- Power: $[\text{length}]^2 * [\text{mass}] / [\text{time}]^3$ (watt, GW)
- Force: $[\text{length}] * [\text{mass}] / [\text{time}]^2$ (newton)

13.11 See Also

- `database_schema` - v4.0 schema reference
- `configuration` - Configuration file format
- `temoa/model_checking/unit_checking/` - Source code
- `tests/test_unit_checking.py` - Unit tests with examples

CHAPTER
FOURTEEN

REFERENCES

BIBLIOGRAPHY

- [open_source_realities] The two main goals behind Temoa are transparency and repeatability, hence the MIT 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.
- [efficiency_table] 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.
- [glpk_presolve] Circa 2013, GLPK uses more memory than commercial alternatives and has vastly weaker presolve capabilities.
- [esom_definition] For a more in-depth description of energy system optimization models (ESOMs) and guidance on how to use them, please see: DeCarolís et al. (2017) “Formalizing best practice for energy system optimization modelling”, *Applied Energy*, 194: 184-198.
- [web_browser_svg] SVG support in web browsers is currently hit or miss. The most recent versions of Chromium, Google Chrome, and Mozilla Firefox support SVG well enough for Temoa’s current use of SVG.
- [return_expression] 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.
- [abstract_model] 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.
- [BrookeRosenthal03] Anthony Brooke and Richard E. Rosenthal. *GAMS*. GAMS Development, 2003.
- [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.