# Contents

1 User guide
   1.1 Introduction ................................................................. 3
   1.2 Download and installation ............................................. 5
   1.3 Building a model ......................................................... 7
   1.4 Running a model ......................................................... 12
   1.5 Analysing a model ...................................................... 17
   1.6 Tutorials ................................................................. 20
   1.7 More info ............................................................... 34
   1.8 Development guide ..................................................... 88

2 API documentation
   2.1 API Documentation .................................................... 91
   2.2 Index ................................................................. 98

3 Release history
   3.1 Release History ......................................................... 99

4 License

Bibliography

Python Module Index
v0.6.3 (Release history)

This is the documentation for version 0.6.3. See the main project website for contact details and other useful information.

Calliope focuses on flexibility, high spatial and temporal resolution, the ability to execute many runs based on the same base model, and a clear separation of framework (code) and model (data). Its primary focus is on planning energy systems at scales ranging from urban districts to entire continents. In an optional operational mode it can also test a pre-defined system under different operational conditions.

A model based on Calliope consists of a collection of text files (in YAML and CSV formats) that define the technologies, locations and resource potentials. Calliope takes these files, constructs an optimisation problem, solves it, and reports results in the form of xarray Datasets which in turn can easily be converted into Pandas data structures, for easy analysis with Calliope’s built-in tools or the standard Python data analysis stack.

Calliope’s built-in tools allow interactive exploration of results, as shown in the following example of a model that includes three energy carriers (electricity, heat, and gas):

Calliope is developed in the open on GitHub and contributions are very welcome (see the Development guide). See the list of open issues and planned milestones for an overview of where development is heading, and join us on Gitter to ask questions or discuss code.

Key features of Calliope include:

- Model specification in an easy-to-read and machine-processable YAML format
- Generic technology definition allows modelling any mix of production, storage and consumption
- Resolved in space: define locations with individual resource potentials
- Resolved in time: read time series with arbitrary resolution
- Able to run on high-performance computing (HPC) clusters
- Uses a state-of-the-art Python toolchain based on Pyomo, xarray, and Pandas
- Freely available under the Apache 2.0 license
1.1 Introduction

The basic process of modelling with Calliope is based on three steps:

1. Create a model from scratch or by adjusting an existing model *(Building a model)*
2. Run your model *(Running a model)*
3. Analyse and visualise model results *(Analysing a model)*

1.1.1 Energy system models

Energy system models allow analysts to form internally coherent scenarios of how energy is extracted, converted, transported, and used, and how these processes might change in the future. These models have been gaining renewed importance as methods to help navigate the climate policy-driven transformation of the energy system.

Calliope is an attempt to design an energy system model from the ground up with specific design goals in mind (see below). Therefore, the model approach and data format layout may be different from approaches used in other models. The design of the nodes approach used in Calliope was influenced by the power nodes modelling framework by [Heussen2010], but Calliope is different from traditional power system modelling tools, and does not provide features such as power flow analysis.

Calliope was designed to address questions around the transition to renewable energy, so there are tools that are likely to be more suitable for other types of questions. In particular, the following related energy modelling systems are available under open source or free software licenses:

- **SWITCH**: A power system model focused on renewables integration, using multi-stage stochastic linear optimisation, as well as hourly resource potential and demand data. Written in the commercial AMPL language and GPL-licensed [Fripp2012].
- **Temoa**: An energy system model with multi-stage stochastic optimisation functionality which can be deployed to computing clusters, to address parametric uncertainty. Written in Python/Pyomo and AGPL-licensed [Hunter2013].
• **OSeMOSYS**: A simplified energy system model similar to the MARKAL/TIMES model families, which can be used as a stand-alone tool or integrated in the LEAP energy model. Written in GLPK, a free subset of the commercial AMPL language, and Apache 2.0-licensed [Howells2011].

Additional energy models that are partially or fully open can be found on the Open Energy Modelling Initiative’s wiki.

### 1.1.2 Rationale

Calliope was designed with the following goals in mind:

- Designed from the ground up to analyze energy systems with high shares of renewable energy or other variable generation
- Formulated to allow arbitrary spatial and temporal resolution, and equipped with the necessary tools to deal with time series input data
- Allow easy separation of model code and data, and modular extensibility of model code
- Make models easily modifiable, archiveable and auditable (e.g. in a Git repository), by using well-defined and human-readable text formats
- Simplify the definition and deployment of large numbers of model runs to high-performance computing clusters
- Able to run stand-alone from the command-line, but also provide an API for programmatic access and embedding in larger analyses
- Be a first-class citizen of the Python world (installable with conda and pip, with properly documented and tested code that mostly conforms to PEP8)
- Have a free and open-source code base under a permissive license

### 1.1.3 Acknowledgments

Initial development was partially funded by the Grantham Institute at Imperial College London and the European Institute of Innovation & Technology’s Climate-KIC program.

### 1.1.4 License

Calliope is released under the Apache 2.0 license, which is a permissive open-source license much like the MIT or BSD licenses. This means that Calliope can be incorporated in both commercial and non-commercial projects.

---

**Copyright 2013-2018 Calliope contributors listed in AUTHORS**

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
1.1.5 References

1.2 Download and installation

1.2.1 Requirements

Calliope has been tested on Linux, macOS, and Windows.

Running Calliope requires four things:

1. The Python programming language, version 3.6 or higher.
2. A number of Python add-on modules (see below for the complete list).
3. A solver: Calliope has been tested with GLPK, CBC, Gurobi, and CPLEX. Any other solver that is compatible with Pyomo should also work.
4. The Calliope software itself.

1.2.2 Recommended installation method

The easiest way to get a working Calliope installation is to use the free conda package manager, which can install all of the four things described above in a single step.

To get conda, download and install the “Miniconda” distribution for your operating system (using the version for Python 3).

With Miniconda installed, you can create a new Python 3.6 environment called "calliope" with all the necessary modules, including the free and open source GLPK solver, by running the following command in a terminal or command-line window:

```
$ conda create -c conda-forge -n calliope python=3.6 calliope
```

To use Calliope, you need to activate the calliope environment each time. On Linux and macOS:

```
$ source activate calliope
```

On Windows:

```
$ activate calliope
```

You are now ready to use Calliope together with the free and open source GLPK solver. Read the next section for more information on alternative solvers.

1.2.3 Updating an existing installation

If following the recommended installation method above, the following command, assuming the conda environment is active, will update Calliope to the newest version:

```
$ conda update -c conda-forge calliope
```
1.2.4 Solvers

You need at least one of the solvers supported by Pyomo installed. CPLEX or Gurobi are recommended for large problems, and have been confirmed to work with Calliope. Refer to the documentation of your solver on how to install it.

GLPK

GLPK is free and open-source, but can take too much time and/or too much memory on larger problems. If using the recommended installation approach above, GLPK is already installed in the calliope environment. To install GLPK manually, refer to the GLPK website.

CBC

CBC is another free and open-source option. CBC can be installed via conda on Linux and macOS by running `conda install -c conda-forge coincbc`. Windows binary packages and further documentation are available at the CBC website.

Gurobi

Gurobi is commercial but significantly faster than GLPK and CBC, which is relevant for larger problems. It needs a license to work, which can be obtained for free for academic use by creating an account on gurobi.com.

While Gurobi can be installed via conda (`conda install -c gurobi gurobi`) we recommend downloading and installing the installer from the Gurobi website, as the conda package has repeatedly shown various issues.

After installing, log on to the Gurobi website and obtain a (free academic or paid commercial) license, then activate it on your system via the instructions given online (using the `grbgetkey` command).

CPLEX

Another commercial alternative is CPLEX. IBM offer academic licenses for CPLEX. Refer to the IBM website for details.

1.2.5 Python module requirements

Refer to `requirements/base.yml` in the Calliope repository for a full and up-to-date listing of required third-party packages.

Some of the key packages Calliope relies on are:

- Pyomo
- Pandas
- Xarray
- Plotly
- Jupyter (optional, but highly recommended, and used for the example notebooks in the tutorials)
1.3 Building a model

In short, a Calliope model works like this: supply technologies can take a resource from outside of the modeled system and turn it into a specific energy carrier in the system. The model specifies one or more locations along with the technologies allowed at those locations. Transmission technologies can move energy of the same carrier from one location to another, while conversion technologies can convert one carrier into another at the same location. Demand technologies remove energy from the system, while storage technologies can store energy at a specific location. Putting all of these possibilities together allows a modeller to specify as simple or as complex a model as necessary to answer a given research question.

In more technical terms, Calliope allows a modeller to define technologies with arbitrary characteristics by “inheriting” basic traits from a number of included base tech groups – supply, supply_plus, demand, conversion, conversion_plus, and transmission. These groups are described in more detail in List of abstract base technology groups.

1.3.1 Terminology

The terminology defined here is used throughout the documentation and the model code and configuration files:

- **Technology**: a technology that produces, consumes, converts or transports energy
- **Location**: a site which can contain multiple technologies and which may contain other locations for energy balancing purposes
- **Resource**: a source or sink of energy that can (or must) be used by a technology to introduce into or remove energy from the system
- **Carrier**: an energy carrier that groups technologies together into the same network, for example electricity or heat.

As more generally in constrained optimisation, the following terms are also used:

- **Parameter**: a fixed coefficient that enters into model equations
- **Variable**: a variable coefficient (decision variable) that enters into model equations
- **Set**: an index in the algebraic formulation of the equations
- **Constraint**: an equality or inequality expression that constrains one or several variables

1.3.2 Files that define a model

Calliope models are defined through YAML files, which are both human-readable and computer-readable, and CSV files (a simple tabular format) for time series data.

It makes sense to collect all files belonging to a model inside a single model directory. The layout of that directory typically looks roughly like this (+ denotes directories, - files):

```
+ example_model
  + model_config
    - locations.yaml
    - techs.yaml
  + timeseries_data
    - solar_resource.csv
    - electricity_demand.csv
  - model.yaml
  - scenarios.yaml
```
In the above example, the files `model.yaml`, `locations.yaml` and `techs.yaml` together are the model definition. This definition could be in one file, but it is more readable when split into multiple. We use the above layout in the example models and in our research!

Inside the `timeseries_data` directory, timeseries are stored as CSV files. The location of this directory can be specified in the model configuration, e.g. in `model.yaml`.

**Note:** The easiest way to create a new model is to use the `calliope new` command, which makes a copy of one of the built-in examples models:

```bash
$ calliope new my_new_model
```

This creates a new directory, `my_new_model`, in the current working directory.

By default, `calliope new` uses the national-scale example model as a template. To use a different template, you can specify the example model to use, e.g.: `--template=urban_scale`.

See also:

* YAML configuration file format*, *Built-in example models*, *Time series data*

### 1.3.3 Model configuration (model)

The model configuration specifies all aspects of the model to run. It is structured into several top-level headings (keys in the YAML file): `model`, `techs`, `locations`, `links`, and `run`. We will discuss each of these in turn, starting with `model`:

```yaml
model:
  name: 'My energy model'
  timeseries_data_path: 'timeseries_data'
  reserve_margin:
    power: 0
    subset_time: ['2005-01-01', '2005-01-05']
```

Besides the model’s name (`name`) and the path for CSV time series data (`timeseries_data_path`), model-wide constraints can be set, like `reserve_margin`.

To speed up model runs, the above example specifies a time subset to run the model over only five days of time series data (`subset_time: ['2005-01-01', '2005-01-05']`)—this is entirely optional. Usually, a full model will contain at least one year of data, but subsetting time can be useful to speed up a model for testing purposes.

See also:

* National scale example model*, *List of model settings*

### 1.3.4 Technologies (techs)

The `techs` section in the model configuration specifies all of the model’s technologies. In our current example, this is in a separate file, `model_config/techs.yaml`, which is imported into the main `model.yaml` file alongside the file for locations described further below:

```yaml
import:
  - 'model_config/techs.yaml'
  - 'model_config/locations.yaml'
```
Note: The import statement can specify a list of paths to additional files to import (the imported files, in turn, may include further files, so arbitrary degrees of nested configurations are possible). The import statement can either give an absolute path or a path relative to the importing file.

The following example shows the definition of a ccgt technology, i.e. a combined cycle gas turbine that delivers electricity:

```plaintext
ccgt:
  essentials:
    name: 'Combined cycle gas turbine'
    color: '#FDC97D'
    parent: supply
    carrier_out: power
  constraints:
    resource: inf
    energy_eff: 0.5
    energy_cap_max: 40000  # kW
    energy_cap_max_systemwide: 100000  # kW
    energy_ramping: 0.8
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap: 750  # USD per kW
      om_con: 0.02  # USD per kWh
```

Each technology must specify some essentials, most importantly a name, the abstract base technology it is inheriting from (parent), and its energy carrier (carrier_out in the case of a supply technology). Specifying a color is optional but useful for using the built-in visualisation tools (see Analysing a model).

The constraints section gives all constraints for the technology, such as allowed capacities, conversion efficiencies, the life time (used in levelised cost calculations), and the resource it consumes (in the above example, the resource is set to infinite via inf).

The costs section gives costs for the technology. Calliope uses the concept of “cost classes” to allow accounting for more than just monetary costs. The above example specifies only the monetary cost class, but any number of other classes could be used, for example co2 to account for emissions.

By default the monetary cost class is used in the objective function, which seeks to minimize total costs. Additional cost classes can be created simply by adding them to the definition of costs for a technology. To use an alternative cost class and/or sense (minimize/maximize) in the objective function, the objective_options parameter can be set in the run configuration, e.g. `objective_options: {'cost_class': 'emissions', 'sense': 'minimize'}`.

See also:

List of possible constraints, List of possible costs, tutorials, built-in examples

Allowing for unmet demand

For a model to find a feasible solution, supply must always be able to meet demand. To avoid the solver failing to find a solution, you can ensure feasibility:

```plaintext
run:
    ensure_feasibility: true
```
This will create an unmet_demand decision variable in the optimisation, which can pick up any mismatch between supply and demand, across all energy carriers. It has a very high cost associated with its use, so it will only appear when absolutely necessary.

Note: When ensuring feasibility, you can also set a big M value (run.bigM). This is the “cost” of unmet demand. It is possible to make model convergence very slow if bigM is set too high. default bigM is 1x10^9, but should be close to the maximum total system cost that you can imagine. This is perhaps closer to 1x10^6 for urban scale models.

1.3.5 Locations and links (locations, links)

A model can specify any number of locations. These locations are linked together by transmission technologies. By consuming an energy carrier in one location and outputting it in another, linked location, transmission technologies allow resources to be drawn from the system at a different location from where they are brought into it.

The locations section specifies each location:

```ini
locations:
  region1:
    coordinates: {lat: 40, lon: -2}
    techs:
      unmet_demand_power:
      demand_power:
      ccgt:
        constraints:
          energy_cap_max: 30000
```

Locations can optionally specify coordinates (used in visualisation or to compute distance between them) and must specify techs allowed at that location. As seen in the example above, each allowed tech must be listed, and can optionally specify additional location-specific parameters (constraints or costs). If given, location-specific parameters supersede any model-wide constraints a technology defines in the techs section for that location.

The links section specifies possible transmission links between locations in the form location1,location2:

```ini
links:
  region1,region2:
    techs:
      ac_transmission:
        constraints:
          energy_cap_max: 10000
        costs.monetary:
          energy_cap: 100
```

In the above example, an high-voltage AC transmission line is specified to connect region1 with region2. For this to work, a transmission technology called ac_transmission must have previously been defined in the model's techs section. There, it can be given model-wide constraints or costs. As in the case of locations, the links section can specify per-link parameters (constraints or costs) that supersede any model-wide parameters.

The modeller can also specify a distance for each link, and use per-distance constraints and costs for transmission technologies.

See also:

List of possible constraints, List of possible costs.
1.3.6 Run configuration (run)

The only required setting in the run configuration is the solver to use:

```yaml
run:
  solver: glpk
  model: plan
```

The most important parts of the run section are solver and mode. A model can run either in planning mode (plan) or operational mode (operate). In planning mode, capacities are determined by the model, whereas in operational mode, capacities are fixed and the system is operated with a receding horizon control algorithm.

Possible options for solver include glpk, gurobi, cplex, and cbc. The interface to these solvers is done through the Pyomo library. Any solver compatible with Pyomo should work with Calliope.

For solvers with which Pyomo provides more than one way to interface, the additional solver_io option can be used. In the case of Gurobi, for example, it is usually fastest to use the direct Python interface:

```yaml
run:
  solver: gurobi
  solver_io: python
```

Note: The opposite is currently true for CPLEX, which runs faster with the default solver_io.

Further optional settings, including debug settings, can be specified in the run configuration.

See also:
List of run settings, Debugging failing runs, Solver options, documentation on operational mode.

1.3.7 Scenarios and overrides

To make it easier to run a given model multiple times with slightly changed settings or constraints, for example, varying the cost of a key technology, it is possible to define and apply scenarios and overrides. “Overrides” are blocks of YAML that specify configurations that expand or override parts of the base model. “Scenarios” are combinations of any number of such overrides. Both are specified at the top level of the model configuration, as in this example model.yaml file:

```yaml
scenarios:
  high_cost_2005: "high_cost,year2005"
  high_cost_2006: "high_cost,year2006"

overrides:
  high_cost:
    techs.onshore_wind.costs.monetary.energy_cap: 2000
    year2005:
      model.subset_time: ['2005-01-01', '2005-12-31']
    year2006:
      model.subset_time: ['2006-01-01', '2006-12-31']

model:
  ...

run:
  ...
```
Each override is given by a name (e.g. `high_cost`) and any number of model settings – anything in the model configuration can be overridden by an override. In the above example, one override defines higher costs for an `onshore_wind` tech while the two other overrides specify different time subsets, so would run an otherwise identical model over two different periods of time series data.

One or several overrides can be applied when running a model, as described in *Running a model*. Overrides can also be combined into scenarios to make applying them at run-time easier. Scenarios consist of a name and a string of comma-delimited override names which together form that scenario.

Scenarios and overrides can be used to generate scripts that run a single Calliope model many times, either sequentially, or in parallel on a high-performance cluster (see *Generating scripts to run a model many times*).

**Note:** Overrides can also import other files. This can be useful if many overrides are defined which share large parts of model configuration, such as different levels of interconnection between model zones. See *Imports in overrides* for details.

See also:

*Generating scripts to run a model many times*, *Imports in overrides*

### 1.4 Running a model

There are essentially three ways to run a Calliope model:

1. With the `calliope run` command-line tool.
2. By programmatically creating and running a model from within other Python code, or in an interactive Python session.
3. By generating and then executing scripts with the `calliope generate_runs` command-line tool, which is primarily designed for running many scenarios on a high-performance cluster.

#### 1.4.1 Running with the command-line tool

We can easily run a model after creating it (see *Building a model*), saving results to a single NetCDF file for further processing:

```
$ calliope run testmodel/model.yaml --save_netcdf=results.nc
```

The `calliope run` command takes the following options:

- `--save_netcdf={filename.nc}`: Save complete model, including results, to the given NetCDF file. This is the recommended way to save model input and output data into a single file, as it preserves all data fully, and allows later reconstruction of the Calliope model for further analysis.
- `--save_csv={directory name}`: Save results as a set of CSV files to the given directory. This can be handy if the modeler needs results in a simple text-based format for further processing with a tool like Microsoft Excel.
- `--save_plots={filename.html}`: Save interactive plots to the given HTML file (see *Analysing a model* for further details on the plotting functionality).
- `--debug`: Run in debug mode, which prints more internal information, and is useful when troubleshooting failing models.
• --scenario={scenario} and --override_dict={yaml_string}: Specify a scenario, or one or several overrides, to apply to the model, or apply specific overrides from a YAML string (see below for more information)
• --help: Show all available options.

Multiple options can be specified, for example, saving NetCDF, CSV, and HTML plots simultaneously:

```
$ calliope run testmodel/model.yaml --save_netcdf=results.nc --save_csv=outputs --save_plots=plots.html
```

**Warning:** Unlike in versions prior to 0.6.0, the command-line tool in Calliope 0.6.0 and upward does not save results by default – the modeller must specify one of the -save options.

### Applying a scenario or override

The --scenario can be used in three different ways:

- It can be given the name of a scenario defined in the model configuration, as in --scenario=my_scenario
- It can be given the name of a single override defined in the model configuration, as in --scenario=my_override
- It can be given a comma-separated string of several overrides defined in the model configuration, as in --scenario=my_override_1,my_override_2

In the latter two cases, the given override(s) is used to implicitly create a “scenario” on-the-fly when running the model. This allows quick experimentation with different overrides without explicitly defining a scenario combining them.

Assuming we have specified an override called milp in our model configuration, we can apply it to our model with:

```
$ calliope run testmodel/model.yaml --scenario=milp --save_netcdf=results.nc
```

Note that if both a scenario and an override with the same name, such as milp in the above example, exist, Calliope will raise an error, as it will not be clear which one the user wishes to apply.

It is also possible to use the --override_dict option to pass a YAML string that will be applied after anything applied through --scenario:

```
$ calliope run testmodel/model.yaml --override_dict="{'model.subset_time': ['2005-01-01', '2005-01-31']}" --save_netcdf=results.nc
```

See also:

*Analysing a model*, *Scenarios and overrides*

### 1.4.2 Running interactively with Python

The most basic way to run a model programmatically from within a Python interpreter is to create a `Model` instance with a given `model.yaml` configuration file, and then call its `run()` method:

```python
import calliope
model = calliope.Model('path/to/model.yaml')
model.run()
```

1.4. Running a model
Note: If `config` is not specified (i.e. `model = Model()`), an error is raised. See Built-in example models for information on instantiating a simple example model without specifying a custom model configuration.

Note: Calliope logs useful progress information to the INFO log level, but does not change the default log level from WARNING. To see progress information when running interactively, call `calliope.set_log_level('INFO')` immediately after importing Calliope.

Other ways to load a model interactively are:

- Passing an AttrDict or standard Python dictionary to the Model constructor, with the same nested format as the YAML model configuration (top-level keys: model, run, locations, techs).
- Loading a previously saved model from a NetCDF file with `model = calliope.read_netcdf('path/to/saved_model.nc')`. This can either be a pre-processed model saved before its run method was called, which will include input data only, or a completely solved model, which will include input and result data.

After instantiating the Model object, and before calling the `run()` method, it is possible to manually inspect and adjust the configuration of the model. The pre-processed inputs are all held in the xarray Dataset `model.inputs`.

After the model has been solved, an xarray Dataset containing results (`model.results`) can be accessed. At this point, the model can be saved with either `to_csv()` or `to_netcdf()`, which saves all inputs and results, and is equivalent to the corresponding --save options of the command-line tool.

See also:

An example of interactive running in a Python session, which also demonstrates some of the analysis possibilities after running a model, is given in the tutorials. You can download and run the embedded notebooks on your own machine (if both Calliope and the Jupyter Notebook are installed).

Scenarios and overrides

There are two ways to override a base model when running interactively, analogously to the use of the command-line tool (see Applying a scenario or override above):

1. By setting the `scenario` argument, e.g.:
   ```python
def run_scenario(model, scenario):
    model = calliope.Model('model.yaml', scenario=scenario)
```

2. By passing the `override_dict` argument, which is a Python dictionary, an AttrDict, or a YAML string of overrides:
   ```python
   model = calliope.Model('model.yaml', override_dict={
       'run.solver': 'gurobi'
   })
```

Tracking progress

When running Calliope in command line, logging of model pre-processing and solving occurs automatically. Interactively, for example in a Jupyter notebook, you can enable verbose logging by running the following code before instantiating and running a Calliope model:
```python
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(levelname)s: %(message)s',
)

logger = logging.getLogger()
```

This will include model processing output, as well as the output of the chosen solver.

### 1.4.3 Generating scripts for many model runs

Scripts to simplify the creation and execution of a large number of Calliope model runs are generated with the `calliope generate_runs` command-line tool. More detail on this is available in *Generating scripts to run a model many times*.

### 1.4.4 Improving solution times

Large models will take time to solve. The most basic advice is to just let it run on a remote device (another computer or a high performance computing cluster) and forget about it until it is done. However, if you need results *now*, there are ways to improve solution time, invariably at the expense of model ‘accuracy’.

**Number of variables**

The sets `locs`, `techs`, `timesteps`, `carriers`, and `costs` all contribute to model complexity. A reduction of any of these sets will reduce the number of resulting decision variables in the optimisation, which in turn will improve solution times.

**Note:** By reducing the number of locations (e.g. merging nearby locations) you also remove the technologies linking those locations to the rest of the system, which is additionally beneficial.

Currently, we only provide automatic set reduction for timesteps. Timesteps can be resampled (e.g. 1hr -> 2hr intervals), masked (e.g. 1hr -> 12hr intervals except one week of particular interest), or clustered (e.g. 365 days to 5 days, each representing 73 days of the year, with 1hr resolution). In so doing, significant solution time improvements can be achieved.

**See also:**


**Complex technologies**

Calliope is primarily an LP framework, but application of certain constraints will trigger binary or integer decision variables. When triggered, a MILP model will be created.

In both cases, there will be a time penalty, as linear programming solvers are less able to converge on solutions of problems which include binary or integer decision variables. But, the additional functionality can be useful. A purchasing cost allows for a cost curve of the form \( y = Mx + C \) to be applied to a technology, instead of the LP
costs which are all of the form $y = Mx$. Integer units also trigger per-timestep decision variables, which allow technologies to be “on” or “off” at each timestep.

Additionally, in LP models, interactions between timesteps (in storage technologies) can lead to longer solution time. The exact extent of this is as-yet untested.

**Model mode**

Solution time increases more than linearly with the number of decision variables. As it splits the model into ~daily chunks, operational mode can help to alleviate solution time of big problems. This is clearly at the expense of fixing technology capacities. However, one solution is to use a heavily time clustered plan mode to get indicative model capacities. Then run operate mode with these capacities to get a higher resolution operation strategy. If necessary, this process could be iterated.

See also:

*Operational mode*

**Solver choice**

The open-source solvers (GLPK and CBC) are slower than the commercial solvers. If you are an academic researcher, it is recommended to acquire a free licence for Gurobi or CPLEX to very quickly improve solution times. GLPK in particular is slow when solving MILP models. CBC is an improvement, but can still be several orders of magnitude slower at reaching a solution than Gurobi or CPLEX.

We tested solution time for various solver choices on our example models, extended to run over a full year (8760 hours). These runs took place on the University of Cambridge high performance computing cluster, with a maximum run time of 5 hours. As can be seen, CBC is far superior to GLPK. If introducing binary constraints, although CBC is an improvement on GLPK, access to a commercial solver is preferable.

**National scale example model size**

- Variables: 420526 [Nneg: 219026, Free: 105140, Other: 96360]
- Linear constraints: 586972 [Less: 315373, Greater: 10, Equal: 271589]

**MILP urban scale example model**


### Solution time

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLPK</td>
<td>4:35:40</td>
</tr>
<tr>
<td>CBC</td>
<td>0:04:45</td>
</tr>
<tr>
<td>Gurobi (1 thread)</td>
<td>0:02:08</td>
</tr>
<tr>
<td>CPLEX (1 thread)</td>
<td>0:04:55</td>
</tr>
<tr>
<td>Gurobi (4 thread)</td>
<td>0:02:27</td>
</tr>
<tr>
<td>CPLEX (4 thread)</td>
<td>0:02:16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLPK</td>
<td>&gt;5hrs</td>
</tr>
<tr>
<td>CBC</td>
<td>0:52:13</td>
</tr>
<tr>
<td>Gurobi (1 thread)</td>
<td>0:03:21</td>
</tr>
<tr>
<td>CPLEX (1 thread)</td>
<td>0:05:56</td>
</tr>
<tr>
<td>Gurobi (4 thread)</td>
<td>0:03:08</td>
</tr>
<tr>
<td>CPLEX (4 thread)</td>
<td>0:03:26</td>
</tr>
</tbody>
</table>

See also:

*Solver options*
Rerunning a model

After running, if there is an infeasibility you want to address, or simply a few values you don't think were quite right, you can change them and rerun your model. If you change them in `model.inputs`, just rerun the model as `model.run(force_rerun=True)`.

**Note:** `model.run(force_rerun=True)` will replace your current `model.results` and rebuild the entire model backend. You may want to save your model before doing this.

Particularly if your problem is large, you may not want to rebuild the backend to change a few small values. Instead you can interface directly with the backend using the `model.backend` functions, to update individual parameter values and switch constraints on/off. By rerunning the backend specifically, you can optimise your problem with these backend changes, without rebuilding the backend entirely.

**Note:** `model.inputs` and `model.results` will not be changed when updating and rerunning the backend. Instead, a new xarray Dataset is returned.

See also:

*Interfacing with the solver backend*

### 1.4.5 Debugging failing runs

What will typically go wrong, in order of decreasing likelihood:

- The model is improperly defined or missing data. Calliope will attempt to diagnose some common errors and raise an appropriate error message.
- The model is consistent and properly defined but infeasible. Calliope will be able to construct the model and pass it on to the solver, but the solver (after a potentially long time) will abort with a message stating that the model is infeasible.
- There is a bug in Calliope causing the model to crash either before being passed to the solver, or after the solver has completed and when results are passed back to Calliope.

Calliope provides help in diagnosing model issues. See the section on *debugging failing runs*.

### 1.5 Analysing a model

Calliope inputs and results are designed for easy handling. Whatever software you prefer to use for data processing, either the NetCDF or CSV output options should provide a path to importing your Calliope results. If you prefer to not worry about writing your own scripts, then we have that covered too! The built-in plotting functions in `plot` are built on Plotly's interactive visualisation tools to bring your data to life.

#### 1.5.1 Accessing model data and results

A model which solved successfully has two primary Datasets with data of interest:

- `model.inputs`: contains all input data, such as renewable resource capacity factors
- `model.results`: contains all results data, such as dispatch decisions and installed capacities
In both of these, variables are indexed over concatenated sets of locations and technologies, over a dimension we call `loc_techs`. For example, if a technology called `boiler` only exists in location `X1` and not in locations `X2` or `X3`, then it will have a single entry in the `loc_techs` dimension called `X1::boiler`. For parameters which also consider different energy carriers, we use a `loc_tech_carrier` dimension, such that we would have, in the case of the prior boiler example, `X1::boiler::heat`.

This concatenated set formulation is memory-efficient but cumbersome to deal with, so the `model.get_formatted_array(name_of_variable)` function can be used to retrieve a DataArray indexed over separate dimensions (any of `techs`, `locs`, `carriers`, `costs`, `timesteps`, depending on the desired variable).

**Note:** On saving to CSV (see the command-line interface documentation), all variables are saved to a single file each, which are always indexed over all dimensions rather than just the concatenated dimensions.

### 1.5.2 Visualising results

In an interactive Python session, there are four primary visualisation functions: `capacity`, `timeseries`, `transmission`, and `summary`. To gain access to result visualisation without the need to interact with Python, the `summary` plot can also be accessed from the command line interface (see below).

Refer to the API documentation for the analysis module for an overview of available analysis functionality.

Refer to the tutorials for some basic analysis techniques.

#### Plotting time series

The following example shows a timeseries plot of the built-in urban scale example model:

In Python, we get this function by calling `model.plot.timeseries()`. It includes all relevant timeseries information, from both inputs and results. We can force it to only have particular results in the dropdown menu:

```
# Only inputs or only results
model.plot.timeseries(array='inputs')
model.plot.timeseries(array='results')

# Only consumed resource
model.plot.timeseries(array='resource_con')

# Only consumed resource and 'power' carrier flow
model.plot.timeseries(array=['power', 'resource_con'])
```

The data used to build the plots can also be subset and ordered by using the `subset` argument. This uses `xarray`'s `loc` indexing functionality to access subsets of data:

```
# Only show region1 data (rather than the default, which is a sum of all locations)
model.plot.timeseries(subset={'locs': ['region1']})

# Only show a subset of technologies
model.plot.timeseries(subset={'techs': ['ccgt', 'csp']})

# Assuming our model has three techs, 'ccgt', 'csp', and 'battery',
# specifying 'subset' lets us order them in the stacked barchart
model.plot.timeseries(subset={'techs': ['ccgt', 'battery', 'csp']})
```

When aggregating model timesteps with clustering methods, the timeseries plots are adjusted accordingly (example from the built-in time_clustering example model):
See also:

*API documentation for the analysis module*

### Plotting capacities

The following example shows a capacity plot of the built-in urban scale example model:

Functionality is similar to timeseries, this time called by `model.plot.capacity()`. Here we show capacity limits set at input and chosen capacities at output. Choosing dropdowns and subsetting works in the same way as for timeseries plots.

### Plotting transmission

The following example shows a transmission plot of the built-in urban scale example model:

By calling `model.plot.transmission()` you will see installed links, their capacities (on hover), and the locations of the nodes. This functionality only works if you have physically pinpointed your locations using the `coordinates` key for your location.

The above plot uses Mapbox to overlay our transmission plot on Openstreetmap. By creating an account at Mapbox and acquiring a Mapbox access token, you can also create similar visualisations by giving the token to the plotting function: `model.plot.transmission(mapbox_access_token='your token here')`.

Without the token, the plot will fall back on simple country-level outlines. In this urban scale example, the background is thus just grey (zoom out to see the UK!):

**Note:** If the coordinates were in x and y, not lat and lon, the transmission trace would be given on a cartesian plot.

### Plotting flows

The following example shows an energy flow plot of the built-in urban scale example model:

By calling `model.plot.flows()` you will see a plot similar to `transmission`. However, you can see carrier production at each node and along links, at every timestep (controlled by moving a slider). This functionality only works if you have physically pinpointed your locations using the `coordinates` key for your location. It is possible to look at only a subset of the timesteps in the model using the `timestep_index_subset` argument, or to show only every X timestep (where X is an integer) using the `timestep_cycle` argument.

**Note:** If the timestep dimension is particularly large in your model, you will find this visualisation to be slow. Time subsetting is recommended for such a case.

If you cannot see the carrier production for a technology on hovering, it is likely masked by another technology at the same location or on the same link. Hide the masking technology to get the hover info for the technology below.

### Summary plots

If you want all the data in one place, you can run `model.plot.summary(to_file='path/to/file.html')`, which will build a HTML file of all the interactive plots (maintaining the interactivity) and save it to `path/to/file.html`. This HTML file can be opened in a web browser to show all the plots. This functionality is made available in the command line interface by using the command `--save_plots=filename.html` when running the model.
See an example of such a HTML plot here.

See also:

*Running with the command-line tool*

**Saving publication-quality SVG figures**

On calling any of the three primary plotting functions, you can also set `save_svg=True` for a high quality vector graphic to be saved. This file can be prepared for publication in programs like Inkscape.

Note: For similar results in the command line interface, you'll currently need to save your model to netcdf `--save_netcdf={filename.nc}` then load it into a Calliope Model object in Python. Once there, you can use the above functions to get your SVGs.

### 1.5.3 Reading solutions

Calliope provides functionality to read a previously-saved model from a single NetCDF file:

```python
solved_model = calliope.read_netcdf('my_saved_model.nc')
```

In the above example, the model’s input data will be available under `solved_model.inputs`, while the results (if the model had previously been solved) are available under `solved_model.results`.

Both of these are `xarray.Datasets` and can be further processed with Python.

See also:

The `xarray documentation` should be consulted for further information on dealing with Datasets. Calliope’s NetCDF files follow the `CF conventions` and can easily be processed with any other tool that can deal with NetCDF.

### 1.6 Tutorials

The tutorials are based on the built-in example models, they explain the key steps necessary to set up and run simple models. Refer to the other parts of the documentation for more detailed information on configuring and running more complex models. The built-in examples are simple on purpose, to show the key components of a Calliope model with which models of arbitrary complexity can be built.

The **first tutorial** builds a model for part of a national grid, exhibiting the following Calliope functionality:

- Use of supply, supply_plus, demand, storage and transmission technologies
- Nested locations
- Multiple cost types

The **second tutorial** builds a model for part of a district network, exhibiting the following Calliope functionality:

- Use of supply, demand, conversion, conversion_plus, and transmission technologies
- Use of multiple energy carriers
- Revenue generation, by carrier export

The **third tutorial** extends the second tutorial, exhibiting binary and integer decision variable functionality (extended an LP model to a MILP model)
1.6.1 Tutorial 1: national scale

This example consists of two possible power supply technologies, a power demand at two locations, the possibility for battery storage at one of the locations, and a transmission technology linking the two. The diagram below gives an overview:

Fig. 1: Overview of the built-in national-scale example model

Supply-side technologies

The example model defines two power supply technologies.

The first is \textit{ccgt} (combined-cycle gas turbine), which serves as an example of a simple technology with an infinite resource. Its only constraints are the cost of built capacity (\texttt{energy\_cap}) and a constraint on its maximum built capacity.

Fig. 2: The layout of a supply node, in this case \textit{ccgt}, which has an infinite resource, a carrier conversion efficiency (\texttt{energy\_eff}), and a constraint on its maximum built \texttt{energy\_cap} (which puts an upper limit on \texttt{energy\_prod}).

The definition of this technology in the example model’s configuration looks as follows:

\begin{verbatim}
ccgt:
  essentials:
    name: 'Combined cycle gas turbine'
    color: '#E37A72'
    parent: supply
    carrier_out: power
  constraints:
    resource: inf
    energy_eff: 0.5
    energy_cap_max: 40000  # kW
    energy_cap_max_systemwide: 100000  # kW
    energy_ramping: 0.8
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap: 750  # USD per kW
      om_con: 0.02  # USD per kWh
\end{verbatim}

There are a few things to note. First, \textit{ccgt} defines essential information: a name, a color (given as an HTML color code, for later visualisation), its parent, \texttt{supply}, and its \texttt{carrier\_out, power}. It has set itself up as a power supply technology. This is followed by the definition of constraints and costs (the only cost class used is monetary, but this is where other “costs”, such as emissions, could be defined).

\textbf{Note:} There are technically no restrictions on the units used in model definitions. Usually, the units will be kW and kWh, alongside a currency like USD for costs. It is the responsibility of the modeler to ensure that units are correct and consistent. Some of the analysis functionality in the \texttt{analysis} module assumes that kW and kWh are used when drawing figure and axis labels, but apart from that, there is nothing preventing the use of other units.

The second technology is \textit{csp} (concentrating solar power), and serves as an example of a complex \texttt{supply\_plus} technology making use of:
• a finite resource based on time series data
• built-in storage
• plant-internal losses (parasitic_eff)

Fig. 3: The layout of a more complex node, in this case csp, which makes use of most node-level functionality available.

This definition in the example model’s configuration is more verbose:

```
csp:
  essentials:
    name: 'Concentrating solar power'
    color: '#F9CF22'
    parent: supply_plus
    carrier_out: power
  constraints:
    storage_cap_max: 614033
    charge_rate: 1
    storage_loss: 0.002
    resource: file=csp_resource.csv
    resource_unit: energy_per_area
    energy_eff: 0.4
    parasitic_eff: 0.9
    resource_area_max: inf
    energy_cap_max: 10000
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      storage_cap: 50
      resource_area: 200
      resource_cap: 200
      energy_cap: 1000
      om_prod: 0.002
```

Again, csp has the definitions for name, color, parent, and carrier_out. Its constraints are more numerous: it defines a maximum storage capacity (storage_cap_max), an hourly storage loss rate (storage_loss), then specifies that its resource should be read from a file (more on that below). It also defines a carrier conversion efficiency of 0.4 and a parasitic efficiency of 0.9 (i.e., an internal loss of 0.1). Finally, the resource collector area and the installed carrier conversion capacity are constrained to a maximum.

The costs are more numerous as well, and include monetary costs for all relevant components along the conversion from resource to carrier (power): storage capacity, resource collector area, resource conversion capacity, energy conversion capacity, and variable operational and maintenance costs. Finally, it also overrides the default value for the monetary interest rate.

**Storage technologies**

The second location allows a limited amount of battery storage to be deployed to better balance the system. This technology is defined as follows:

Fig. 4: A storage node with an energy_eff and storage_loss.
battery:
    essentials:
        name: 'Battery storage'
        color: '#3B61E3'
        parent: storage
carrier: power
constraints:
    energy_cap_max: 1000  # kW
    storage_cap_max: inf
    charge_rate: 4
    energy_eff: 0.95  # 0.95 * 0.95 = 0.9025 round trip efficiency
    storage_loss: 0  # No loss over time assumed
    lifetime: 25
costs:
    monetary:
        interest_rate: 0.10
        storage_cap: 200  # USD per kWh storage capacity

The constraints give a maximum installed generation capacity for battery storage together with a charge rate (charge_rate) of 4, which in turn limits the storage capacity. The charge rate is the charge/discharge rate / storage capacity (a.k.a the battery reservoir). In the case of a storage technology, energy_eff applies twice: on charging and discharging. In addition, storage technologies can lose stored energy over time – in this case, we set this loss to zero.

Other technologies

Three more technologies are needed for a simple model. First, a definition of power demand:

Power demand is a technology like any other. We will associate an actual demand time series with the demand technology later.

What remains to set up is a simple transmission technologies. Transmission technologies (like conversion technologies) look different than other nodes, as they link the carrier at one location to the carrier at another (or, in the case of conversion, one carrier to another at the same location):

(continues on next page)
ac_transmission has an efficiency of 0.85, so a loss during transmission of 0.15, as well as some cost definitions.

free_transmission allows local power transmission from any of the csp facilities to the nearest location. As the name suggests, it applies no cost or efficiency losses to this transmission.

Locations

In order to translate the model requirements shown in this section’s introduction into a model definition, five locations are used: region-1, region-2, region1-1, region1-2, and region1-3.

The technologies are set up in these locations as follows:

Let’s now look at the first location definition:

There are several things to note here:

- The location specifies a dictionary of technologies that it allows (techs), with each key of the dictionary referring to the name of technologies defined in our techs.yaml file. Note that technologies listed here must have been defined elsewhere in the model configuration.
• It also overrides some options for both demand_power and ccgt. For the latter, it simply sets a location-specific maximum capacity constraint. For demand_power, the options set here are related to reading the demand time series from a CSV file. CSV is a simple text-based format that stores tables by comma-separated rows. Note that we did not define any resource option in the definition of the demand_power technology. Instead, this is done directly via a location-specific override. For this location, the file demand-1.csv is loaded and the column demand is taken (the text after the colon). If no column is specified, Calliope will assume that the column name matches the location name region1-1. Note that in Calliope, a supply is positive and a demand is negative, so the stored CSV data will be negative.

• Coordinates are defined by latitude (lat) and longitude (lon), which will be used to calculate distance of transmission lines (unless we specify otherwise later on) and for location-based visualisation.

The remaining location definitions look like this:

```
region2:
  coordinates: {lat: 40, lon: -8}
  techs:
    demand_power:
      constraints:
        resource: file=demand-2.csv:demand
    battery:

region1-1.coordinates: {lat: 41, lon: -2}
region1-2.coordinates: {lat: 39, lon: -1}
region1-3.coordinates: {lat: 39, lon: -2}

region1-1, region1-2, region1-3:
  techs:
    csp:
```

region2 is very similar to region1, except that it does not allow the ccgt technology. The three region1-locations are defined together, except for their location coordinates, i.e. they each get the exact same configuration. They allow only the csp technology, this allows us to model three possible sites for CSP plants.

For transmission technologies, the model also needs to know which locations can be linked, and this is set up in the model configuration as follows:

```
region1,region2:
  techs:
    ac_transmission:
      constraints:
        energy_cap_max: 10000

region1,region1-1:
  techs:
    free_transmission:

region1,region1-2:
  techs:
    free_transmission:

region1,region1-3:
  techs:
    free_transmission:
```

We are able to override constraints for transmission technologies at this point, such as the maximum capacity of the specific region1 to region2 link shown here.
Running the model

We now take you through running the model in a Jupyter notebook, which is included fully below. To download and run the notebook yourself, you can find it here. You will need to have Calliope installed.

1.6.2 Tutorial 2: urban scale

This example consists of two possible sources of electricity, one possible source of heat, and one possible source of simultaneous heat and electricity. There are three locations, each describing a building, with transmission links between them. The diagram below gives an overview:

Fig. 8: Overview of the built-in urban-scale example model

Supply technologies

This example model defines three supply technologies.

The first two are `supply_gas` and `supply_grid_power`, referring to the supply of gas (natural gas) and electricity, respectively, from the national distribution system. These ‘ininitely’ available national commodities can become energy carriers in the system, with the cost of their purchase being considered at supply, not conversion.

Fig. 9: The layout of a simple supply technology, in this case `supply_gas`, which has a resource input and a carrier output. A carrier conversion efficiency \( \text{energy}_{\text{eff}} \) can also be applied (although isn’t considered for our supply technologies in this problem).

The definition of these technologies in the example model’s configuration looks as follows:

```python
supply_grid_power:
    essentials:
        name: 'National grid import'
        color: '#C5ABE3'
        parent: supply
        carrier: electricity
    constraints:
        resource: inf
        energy_cap_max: 2000
        lifetime: 25
    costs:
        monetary:
            interest_rate: 0.10
            energy_cap: 15
            om_con: 0.1 # 10p/kWh electricity price #ppt

supply_gas:
    essentials:
        name: 'Natural gas import'
        color: '#C98AAD'
        parent: supply
        carrier: gas
    constraints:
        resource: inf
        energy_cap_max: 2000
```

(continues on next page)
The final supply technology is \texttt{pv} (solar photovoltaic power), which serves as an inflexible supply technology. It has a time-dependant resource availability, loaded from file, a maximum area over which it can capture its resource (\texttt{resource\_area\_max}) and a requirement that all available resource must be used (\texttt{force\_resource: True}). This emulates the reality of solar technologies: once installed, their production matches the availability of solar energy.

The efficiency of the DC to AC inverter (which occurs after conversion from resource to energy carrier) is considered in \texttt{parasitic\_eff} and the \texttt{resource\_area\_per\_energy\_cap} gives a link between the installed area of solar panels to the installed capacity of those panels (i.e. kWp).

In most cases, domestic PV panels are able to export excess energy to the national grid. We allow this here by specifying an \texttt{export\_carrier}. Revenue for export will be considered on a per-location basis.

The definition of this technology in the example model’s configuration looks as follows:

```yaml
pv:
  essentials:
    name: 'Solar photovoltaic power'
    color: '#F9D956'
    parent: supply_power_plus
  constraints:
    export_carrier: electricity
    resource: file=pv_resource.csv:per_area # Already accounts for panel efficiency - kWh/m2. Source: Renewables.ninja Solar PV Power - Version: 1.1 - License: https://creativecommons.org/licenses/by-nc/4.0/ - Reference: https://doi.org/10.1016/j.energy.2016.08.060
    resource\_unit: energy\_per\_area
    parasitic\_eff: 0.85 # inverter losses
    energy\_cap\_max: 250
    resource\_area\_max: 1500
    force\_resource: true
    resource\_area\_per\_energy\_cap: 7 # 7m2 of panels needed to fit 1kWp of panels
    lifetime: 25
  costs:
    monetary:
      interest\_rate: 0.10
      energy\_cap: 1350
```

Finally, the parent of the PV technology is not \texttt{supply\_plus}, but rather \texttt{supply\_power\_plus}. We use this to show the possibility of an intermediate technology group, which provides the information on the energy carrier (\texttt{electricity}) and the ultimate abstract base technology (\texttt{supply\_plus}):

```yaml
technologies:
  supply\_power\_plus:
    essentials:
      parent: supply\_plus
      carrier: electricity
```

Intermediate technology groups allow us to avoid repetition of technology information, be it in \texttt{essentials}, \texttt{constraints}, or \texttt{costs}, by linking multiple technologies to the same intermediate group.
Conversion technologies

The example model defines two conversion technologies.

The first is boiler (natural gas boiler), which serves as an example of a simple conversion technology with one input carrier and one output carrier. Its only constraints are the cost of built capacity (costs.monetary.energy_cap), a constraint on its maximum built capacity (constraints.energy_cap.max), and an energy conversion efficiency (energy_eff).

Fig. 10: The layout of a simple node, in this case boiler, which has one carrier input, one carrier output, a carrier conversion efficiency (energy_eff), and a constraint on its maximum built energy_cap (which puts an upper limit on carrier_prod).

The definition of this technology in the example model’s configuration looks as follows:

```plaintext
boiler:
  essentials:
    name: 'Natural gas boiler'
    color: '#8E2999'
    parent: conversion
    carrier_out: heat
    carrier_in: gas
  constraints:
    energy_cap_max: 600
    energy_eff: 0.85
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      om_con: 0.004 # .4p/kWh
```

There are a few things to note. First, boiler defines a name, a color (given as an HTML color code), and a stack_weight. These are used by the built-in analysis tools when analyzing model results. Second, it specifies its parent, conversion, its carrier_in gas, and its carrier_out heat, thus setting itself up as a gas to heat conversion technology. This is followed by the definition of constraints and costs (the only cost class used is monetary, but this is where other “costs”, such as emissions, could be defined).

The second technology is chp (combined heat and power), and serves as an example of a possible conversion_plus technology making use of two output carriers.

Fig. 11: The layout of a more complex node, in this case chp, which makes use of multiple output carriers.

This definition in the example model’s configuration is more verbose:

```plaintext
chp:
  essentials:
    name: 'Combined heat and power'
    color: '#E4AB97'
    parent: conversion_plus
    primary_carrier_out: electricity
    carrier_in: gas
    carrier_out: electricity
    carrier_out_2: heat
  constraints:
    export_carrier: electricity
```

(continues on next page)
energy_cap_max: 1500
energy_eff: 0.405
carrier_ratios.carrier_out_2.heat: 0.8
lifetime: 25
costs:
  monetary:
    interest_rate: 0.10
    energy_cap: 750
    om_prod: 0.004  # .4p/kWh for 4500 operating hours/year
  export: file=export_power.csv

See also:

The conversion_plus tech

Again, chp has the definitions for name, color, parent, and carrier_in/out. It now has an additional carrier (carrier_out_2) defined in its essential information, allowing a second carrier to be produced at the same time as the first carrier (carrier_out). The carrier ratio constraint tells us the ratio of carrier_out_2 to carrier_out that we can achieve, in this case 0.8 units of heat are produced every time a unit of electricity is produced. To produce these units of energy, gas is consumed at a rate of carrier_prod(carrier_out) / energy_eff, so gas consumption is only a function of power output.

As with the pv, the chp an export electricity. The revenue gained from this export is given in the file export_power.csv, in which negative values are given per time step.

Demand technologies

Electricity and heat demand are defined here:

demand_electricity:
  essentials:
    name: 'Electrical demand'
    color: '#072486'
    parent: demand
    carrier: electricity

demand_heat:
  essentials:
    name: 'Heat demand'
    color: '#660507'
    parent: demand
    carrier: heat

Electricity and heat demand are technologies like any other. We will associate an actual demand time series with each demand technology later.

Transmission technologies

In this district, electricity and heat can be distributed between locations. Gas is made available in each location without consideration of transmission.

Fig. 12: A simple transmission node with an energy_eff.
power_lines:
  essentials:
    name: 'Electrical power distribution'
    color: '#6783E3'
    parent: transmission
    carrier: electricity
  constraints:
    energy_cap_max: 2000
    energy_eff: 0.98
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap_per_distance: 0.01

heat_pipes:
  essentials:
    name: 'District heat distribution'
    color: '#823739'
    parent: transmission
    carrier: heat
  constraints:
    energy_cap_max: 2000
    energy_eff_per_distance: 0.975
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap_per_distance: 0.3

power_lines has an efficiency of 0.95, so a loss during transmission of 0.05. heat_pipes has a loss rate per unit distance of 2.5%/unit distance (or energy_eff_per_distance of 97.5%). Over the distance between the two locations of 0.5km (0.5 units of distance), this translates to $2.5^{0.5} = 1.58\%$ loss rate.

## Locations

In order to translate the model requirements shown in this section’s introduction into a model definition, four locations are used: X1, X2, X3, and N1.

The technologies are set up in these locations as follows:

![](fig13.png)

Let’s now look at the first location definition:

X1:
  techs:
  chp:
  pv:
  supply_grid_power:
    costs.monetary.energy_cap: 100 # cost of transformers
  supply_gas:
  demand_electricity:
    constraints.resource: file=demand_power.csv
  demand_heat:

(continues on next page)
There are several things to note here:

- The location specifies a dictionary of technologies that it allows (techs), with each key of the dictionary referring to the name of technologies defined in our techs.yaml file. Note that technologies listed here must have been defined elsewhere in the model configuration.

- It also overrides some options for both demand_electricity, demand_heat, and supply_grid_power. For the latter, it simply sets a location-specific cost. For demands, the options set here are related to reading the demand time series from a CSV file. CSV is a simple text-based format that stores tables by comma-separated rows. Note that we did not define any resource option in the definition of these demands. Instead, this is done directly via a location-specific override. For this location, the files demand_heat.csv and demand_power.csv are loaded. As no column is specified (see national scale example model) Calliope will assume that the column name matches the location name X1. Note that in Calliope, a supply is positive and a demand is negative, so the stored CSV data will be negative.

- Coordinates are defined by cartesian coordinates x and y, which will be used to calculate distance of transmission lines (unless we specify otherwise later on) and for location-based visualisation. These coordinates are abstract, unlike latitude and longitude, and can be used when we don’t know (or care) about the geographical location of our problem.

- An available_area is defined, which will limit the maximum area of all resource_area technologies to the e.g. roof space available at our location. In this case, we just have pv, but the case where solar thermal panels compete with photovoltaic panels for space, this would the sum of the two to the available area.

The remaining location definitions look like this:

X2:
```yaml
  techs:
    boiler:
      costs.monetary.energy_cap: 43.1 # different boiler costs
    pv:
      costs.monetary:
        om_prod: -0.0203 # revenue for just producing electricity
        export: -0.0491 # FIT return for PV export
    supply_gas:
    demand_electricity:
      constraints.resource: file=demand_power.csv
    demand_heat:
      constraints.resource: file=demand_heat.csv
  available_area: 1300
  coordinates: {x: 8, y: 7}
```

X3:
```yaml
  techs:
    boiler:
      costs.monetary.energy_cap: 78 # different boiler costs
    pv:
      constraints:
        energy_cap_max: 50 # changing tariff structure below 50kW
      costs.monetary:
        om_annual: -80.5 # reimbursement per kWp from FIT
    supply_gas:
    demand_electricity:
```
X2 and X3 are very similar to X1, except that they do not connect to the national electricity grid, nor do they contain the chp technology. Specific pv cost structures are also given, emulating e.g. commercial vs. domestic feed-in tariffs.

N1 differs to the others by virtue of containing no technologies. It acts as a branching station for the heat network, allowing connections to one or both of X2 and X3 without double counting the pipeline from X1 to N1. Its definition look like this:

```
N1: # location for branching heat transmission network
    coordinates: {x: 5, y: 7}
```

For transmission technologies, the model also needs to know which locations can be linked, and this is set up in the model configuration as follows:

```
X1,X2:
    techs:
        power_lines:
            distance: 10
X1,X3:
    techs:
        power_lines:
X1,N1:
    techs:
        heat_pipes:
N1,X2:
    techs:
        heat_pipes:
N1,X3:
    techs:
        heat_pipes:
```

The distance measure for the power line is larger than the straight line distance given by the coordinates of X1 and X2, so we can provide more information on non-direct routes for our distribution system. These distances will override any automatic straight-line distances calculated by coordinates.

**Revenue by export**

Defined for both PV and CHP, there is the option to accrue revenue in the system by exporting electricity. This export is considered as a removal of the energy carrier electricity from the system, in exchange for negative cost (i.e. revenue). To allow this, carrier_export: electricity has been given under both technology definitions and an export value given under costs.

The revenue from PV export varies depending on location, emulating the different feed-in tariff structures in the UK for commercial and domestic properties. In domestic properties, the revenue is generated by simply having the installation (per kW installed capacity), as export is not metered. Export is metered in commercial properties, thus revenue is generated directly from export (per kWh exported). The revenue generated by CHP depends on the electricity grid wholesale price per kWh, being 80% of that. These revenue possibilities are reflected in the technologies’ and locations’ definitions.
Running the model

We now take you through running the model in a Jupyter notebook, which is included fully below. To download and run the notebook yourself, you can find it here. You will need to have Calliope installed.

1.6.3 Tutorial 3: Mixed Integer Linear Programming

This example is based on the urban scale example model, but with an override. In the model’s scenarios.yaml file overrides are defined which trigger binary and integer decision variables, creating a MILP model, rather than a conventional LP model.

Units

The capacity of a technology is usually a continuous decision variable, which can be within the range of 0 and energy_cap_max (the maximum capacity of a technology). In this model, we introduce a unit limit on the CHP instead:

```
chp:
  constraints:
    units_max: 4
    energy_cap_per_unit: 300
    energy_cap_min_use: 0.2
  costs:
    monetary:
      energy_cap: 700
      purchase: 40000
```

A unit maximum allows a discrete, integer number of CHP to be purchased, each having a capacity of energy_cap_per_unit. Any of energy_cap_max, energy_cap_min, or energy_cap_equals are now ignored, in favour of units_max, units_min, or units_equals. A useful feature unlocked by introducing this is the ability to set a minimum operating capacity which is only enforced when the technology is operating. In the LP model, energy_cap_min_use would force the technology to operate at least at that proportion of its maximum capacity at each time step. In this model, the newly introduced energy_cap_min_use of 0.2 will ensure that the output of the CHP is 20% of its maximum capacity in any time step in which it has a non-zero output.

Purchase cost

The boiler does not have a unit limit, it still utilises the continuous variable for its capacity. However, we have introduced a purchase cost:

```
boiler:
  costs:
    monetary:
      energy_cap: 35
      purchase: 2000
```

By introducing this, the boiler now has a binary decision variable associated with it, which is 1 if the boiler has a non-zero energy_cap (i.e. the optimisation results in investment in a boiler) and 0 if the capacity is 0. The purchase cost is applied to the binary result, providing a fixed cost on purchase of the technology, irrespective of the technology size. In physical terms, this may be associated with the cost of pipework, land purchase, etc. The purchase cost is also imposed on the CHP, which is applied to the number of integer CHP units in which the solver chooses to invest.
MILP functionality can be easily applied, but convergence is slower as a result of integer/binary variables. It is recommended to use a commercial solver (e.g. Gurobi, CPLEX) if you wish to utilise these variables outside this example model.

Running the model

We now take you through running the model in a Jupyter notebook, which is included fully below. To download and run the notebook yourself, you can find it here. You will need to have Calliope installed.

1.7 More info

This section, as the title suggests, contains more info and more details, and in particular, information on some of Calliope’s more advanced functionality.

We suggest you read the Building a model, Running a model and Analysing a model sections first.

1.7.1 Advanced functionality

Per-distance constraints and costs

Transmission technologies can additionally specify per-distance efficiency (loss) with energy_eff_per_distance and per-distance costs with energy_cap_per_distance:

```
techs:
  my_transmission_tech:
    essentials:
      ...
    constraints:
      # "efficiency" (1-loss) per unit of distance
      energy_eff_per_distance: 0.99
    costs:
      monetary:
        # cost per unit of distance
        energy_cap_per_distance: 10
```

The distance is specified in transmission links:

```
links:
  location1,location2:
    my_transmission_tech:
      distance: 500
      constraints:
        e_cap.max: 10000
```

If no distance is given, but the locations have been given lat and lon coordinates, Calliope will compute distances automatically (based on the length of a straight line connecting the locations).

One-way transmission links

Transmission links are bidirectional by default. To force unidirectionality for a given technology along a given link, you have to set the one_way constraint in the constraint definition of that technology, for that link:
This will only allow transmission from location1 to location2. To swap the direction, the link name must be inverted, i.e. location2,location1.

**Time series data**

**Note:** If a parameter is not explicit in time and space, it can be specified as a single value in the model definition (or, using location-specific definitions, be made spatially explicit). This applies both to parameters that never vary through time (for example, cost of installed capacity) and for those that may be time-varying (for example, a technology’s available resource).

For parameters that vary in time, time series data can be read from CSV files, by specifying `resource: file=filename.csv` to pick the desired CSV file from within the configured timeseries data path (`model.timeseries_data_path`).

By default, Calliope looks for a column in the CSV file with the same name as the location. It is also possible to specify a column too use when setting `resource` per location, by giving the column name with a colon following the filename: `resource: file=filename.csv:column`.

All time series data in a model must be indexed by ISO 8601 compatible time stamps (usually in the format `YYYY-MM-DD hh:mm:ss`, e.g. 2005-01-01 00:00:00), i.e., the first column in the CSV file must be time stamps.

For example, the first few lines of a CSV file giving a resource potential for two locations might look like this:

```
,location1,location2
2005-01-01 00:00:00,0,0
2005-01-01 01:00:00,0,11
2005-01-01 02:00:00,0,18
2005-01-01 03:00:00,0,49
2005-01-01 04:00:00,11,110
2005-01-01 05:00:00,45,300
2005-01-01 06:00:00,90,458
```

**Time resolution adjustment**

Models have a default timestep length (defined implicitly by the timesteps of the model’s time series data). This default resolution can be adjusted over parts of the dataset by specifying time resolution adjustment in the model configuration, for example:

```
model:
  time:
    function: resample
    function_options: {'resolution': '6H'}
```

In the above example, this would resample all time series data to 6-hourly timesteps.

Calliope’s time resolution adjustment functionality allows running a function that can perform arbitrary adjustments to the time series data in the model.
The available options include:

1. Uniform time resolution reduction through the `resample` function, which takes a pandas-compatible rule describing the target resolution (see above example).

2. Deriving representative days from the input time series, by applying the clustering method implemented in `calliope.time.clustering`, for example:

   ```
   model:
   time:
     function: apply_clustering
     function_options:
       clustering_func: kmeans
       how: mean
       k: 20
   ```

When using representative days, a number of additional constraints are added, based on the study undertaken by Kotzur et al. These constraints require a new decision variable `storage_inter_cluster`, which tracks storage between all the dates of the original timeseries. This particular functionality can be disabled by including `storage_inter_cluster: false` in the `function_options` given above.

**Note:** It is also possible to load user-defined representative days, by pointing to a file in `clustering_func` in the same format as pointing to timeseries files in constraints, e.g. `clustering_func: file=clusters.csv:column_name`. Clusters are unique per datestep, so the clustering file is most readable if the index is at datestep resolution. But, the clustering file index can be in timesteps (e.g. if sharing the same file as a constraint timeseries), with the cluster number repeated per timestep in a day. Cluster values should be integer, starting at zero.

3. Heuristic selection of time steps, that is, the application of one or more of the masks defined in `calliope.time.masks`, which will mark areas of the time series to retain at maximum resolution (unmasked) and areas where resolution can be lowered (masked). Options can be passed to the masking functions by specifying `options`. A `time.function` can still be specified and will be applied to the masked areas (i.e. those areas of the time series not selected to remain at the maximum resolution), as in this example, which looks for the week of minimum and maximum potential wind generation (assuming a `wind` technology was specified), then reduces the rest of the input time series to 6-hourly resolution:

   ```
   model:
   time:
     masks:
       - {function: extreme, options: {padding: 'calendar_week', tech: 'wind', how: 'max'}}
       - {function: extreme, options: {padding: 'calendar_week', tech: 'wind', how: 'min'}}
     function: resample
     function_options: {'resolution': '6H'}
   ```

**Warning:** When using time clustering or time masking, the resulting timesteps will be assigned different weights depending on how long a period of time they represent. Weights are used for example to give appropriate weight to the operational costs of aggregated typical days in comparison to individual extreme days, if both exist in the same processed time series. The weighting is accessible in the model data, e.g. through `Model.inputs.timestep_weights`. The interpretation of results when weights are not 1 for all timesteps requires caution. Production values are not scaled according to weights, but costs are multiplied by weight, in order to weight different timesteps appropriately in the objective function. This means that costs and production values are not consistent without manually post-processing them by either multiplying production by weight (production would
then be inconsistent with capacity) or dividing costs by weight. The computation of levelised costs and of capacity factors takes weighting into account, so these values are consistent and can be used as usual.

See also:

See the implementation of constraints in calliope.backend.pyomo.constraints for more detail on timestep weights and how they affect model constraints.

**The supply_plus tech**

The plus tech groups offer complex functionality, for technologies which cannot be described easily. Supply_plus allows a supply technology with internal storage of resource before conversion to the carrier happens. This could be emulated with dummy carriers and a combination of supply, storage, and conversion techs, but the supply_plus tech allows for concise and mathematically more efficient formulation.

An example use of supply_plus is to define a concentrating solar power (CSP) technology which consumes a solar resource, has built-in thermal storage, and produces electricity. See the national-scale built-in example model for an application of this.

See the listing of supply_plus configuration in the abstract base tech group definitions for the additional constraints that are possible.

**Warning:** When analysing results from supply_plus, care must be taken to correctly account for the losses along the transformation from resource to carrier. For example, charging of storage from the resource may have a resource_eff-associated loss with it, while discharging storage to produce the carrier may have a different loss resulting from a combination of energy_eff and parasitic_eff. Such intermediate conversion losses need to be kept in mind when comparing discharge from storage with carrier_prod in the same time step.

**Cyclic storage**

With storage and supply_plus techs, it is possible to link the storage at either end of the timeseries, using cyclic storage. This allows the user to better represent multiple years by just modelling one year. Cyclic storage is activated by default (to deactivate: run.cyclic_storage: false). As a result, a technology’s initial stored energy at a given location will be equal to its stored energy at the end of the model’s last timestep.

For example, for a model running over a full year at hourly resolution, the initial storage at Jan 1st 00:00:00 will be forced equal to the storage at the end of the timestep Dec 31st 23:00:00. By setting storage_initial for a technology, it is also possible to fix the value in the last timestep. For instance, with run.cyclic_storage: true and a storage_initial of zero, the stored energy must be zero by the end of the time horizon.

Without cyclic storage in place (as was the case prior to v0.6.2), the storage tech can have any amount of stored energy by the end of the timeseries. This may prove useful in some cases, but has less physical meaning than assuming cyclic storage.

**Note:** Cyclic storage also functions when time clustering, if allowing storage to be tracked between clusters (see Time resolution adjustment). However, it cannot be used in operate run mode.
The conversion_plus tech

The plus tech groups offer complex functionality, for technologies which cannot be described easily. Conversion_plus allows several carriers to be converted to several other carriers. Describing such a technology requires that the user understands the carrier_ratios, i.e. the interactions and relative efficiencies of carrier inputs and outputs.

The conversion_plus technologies allows for up to three carrier groups as inputs (carrier_in, carrier_in_2 and carrier_in_3) and up to three carrier groups as outputs (carrier_out, carrier_out_2 and carrier_out_3). A carrier group can contain any number of carriers.

The efficiency of a conversion_plus tech dictates how many units of carrier_out are produced per unit of consumed carrier_in. A unit of carrier_out_2 and of carrier_out_3 is produced each time a unit of carrier_out is produced. Similarly, a unit of Carrier_in_2 and of carrier_in_3 is consumed each time a unit of carrier_in is consumed. Within a given carrier group (e.g. carrier_out_2) any number of carriers can meet this one unit. The carrier_ratio of any carrier compares it either to the production of one unit of carrier_out or to the consumption of one unit of carrier_in.

In this section, we give examples of a few conversion_plus technologies alongside the YAML formulation required to construct them:

Combined heat and power

A combined heat and power plant produces electricity, in this case from natural gas. Waste heat that is produced can be used to meet nearby heat demand (e.g. via district heating network). For every unit of electricity produced, 0.8 units of heat are always produced. This is analogous to the heat to power ratio (HTP). Here, the HTP is 0.8.

```
chp:
  essentials:
    name: Combined heat and power
    carrier_in: gas
    carrier_out: electricity
    carrier_out_2: heat
    primary_carrier_out: electricity
  constraints:
    energy_eff: 0.45
    energy_cap_max: 100
    carrier_ratios.carrier_out_2.heat: 0.8
```

Air source heat pump

The output energy from the heat pump can be either heat or cooling, simulating a heat pump that can be useful in both summer and winter. For each unit of electricity input, one unit of output is produced. Within this one unit of carrier_out, there can be a combination of heat and cooling. Heat is produced with a COP of 5, cooling with a COP of 3. If only heat were produced in a timestep, 5 units of it would be available in carrier_out; similarly 3 units for cooling. In another timestep, both heat and cooling might be produced with e.g. 2.5 units heat + 1.5 units cooling = 1 unit of carrier_out.
ahp:
   essentials:
       name: Air source heat pump
       carrier_in: electricity
       carrier_out: [heat, cooling]
       primary_carrier_out: heat
   constraints:
       energy_eff: 1
       energy_cap_max: 100
       carrier_ratios:
           carrier_out:
               heat: 5
               cooling: 3

Combined cooling, heat and power (CCHP)

A CCHP plant can use generated heat to produce cooling via an absorption chiller. As with the CHP plant, electricity is produced at 45% efficiency. For every unit of electricity produced, 1 unit of carrier_out_2 must be produced, which can be a combination of 0.8 units of heat and 0.5 units of cooling. Some example ways in which the model could decide to operate this unit in a given time step are:

- 1 unit of gas (carrier_in) is converted to 0.45 units of electricity (carrier_out) and (0.8 * 0.45) units of heat (carrier_out_2)
- 1 unit of gas is converted to 0.45 units electricity and (0.5 * 0.45) units of cooling
- 1 unit of gas is converted to 0.45 units electricity, (0.3 * 0.8 * 0.45) units of heat, and (0.7 * 0.5 * 0.45) units of cooling

Advanced gas turbine

This technology can choose to burn methane (CH:sub:4) or send hydrogen (H:sub:2) through a fuel cell to produce electricity. One unit of carrier_in can be met by any combination of methane and hydrogen. If all methane, 0.5 units of carrier_out would be produced for 1 unit of carrier_in (energy_eff). If all hydrogen, 0.25 units of carrier_out would be produced for the same amount of carrier_in (energy_eff * hydrogen carrier ratio).
Complex fictional technology

There are few instances where using the full capacity of a conversion_plus tech is physically possible. Here, we have a fictional technology that combines fossil fuels with biomass/waste to produce heat, cooling, and electricity. Different ‘grades’ of heat can be produced, the higher grades having an alternative. High grade heat (high_T_heat) is produced and can be used directly, or used to produce electricity (via e.g. organic rankine cycle). carrier_out is thus a combination of these two. carrier_out_2 can be 0.3 units mid grade heat for every unit carrier_out or 0.2 units cooling. Finally, 0.1 units carrier_out_3, low grade heat, is produced for every unit of carrier_out.

A primary_carrier_out must be defined when there are multiple carrier_out values defined, similarly primary_carrier_in can be defined for carrier_in. primary_carriers can be defined as any carrier in a technology’s input/output carriers (including secondary and tertiary carriers). The chosen output carrier will be the one to which production costs are applied (reciprocally, input carrier for consumption costs).

Note: Conversion_plus technologies can also export any one of their output carriers, by specifying that carrier as carrier_export.
Revenue and export

It is possible to specify revenues for technologies simply by setting a negative cost value. For example, to consider a feed-in tariff for PV generation, it could be given a negative operational cost equal to the real operational cost minus the level of feed-in tariff received.

Export is an extension of this, allowing an energy carrier to be removed from the system without meeting demand. This is analogous to e.g. domestic PV technologies being able to export excess electricity to the national grid. A cost (or negative cost: revenue) can then be applied to export.

**Note:** Negative costs can be applied to capacity costs, but the user must ensure a capacity limit has been set. Otherwise, optimisation will be unbounded.

Using `tech_groups` to group configuration

In a large model, several very similar technologies may exist, for example, different kinds of PV technologies with slightly different cost data or with different potentials at different model locations.

To make it easier to specify closely related technologies, `tech_groups` can be used to specify configuration shared between multiple technologies. The technologies then give the `tech_group` as their parent, rather than one of the abstract base technologies.

For example:

```plaintext
tech_groups:
  pv:
    essentials:
      parent: supply
      carrier: power
    constraints:
      resource: file=pv_resource.csv
      lifetime: 30
    costs:
      monetary:
        om_annual_investment_fraction: 0.05
        depreciation_rate: 0.15

technologies:
  pv_large_scale:
    essentials:
      parent: pv
      name: 'Large-scale PV'
    constraints:
      energy_cap_max: 2000
    costs:
      monetary:
        e_cap: 750
  pv_rooftop:
    essentials:
      parent: pv
      name: 'Rooftop PV'
    constraints:
      energy_cap_max: 10000
    costs:
      monetary:
        e_cap: 1000
```

1.7. More info
None of the `tech_groups` appear in model results, they are only used to group model configuration values.

### Using the `group_share` constraint

The `group_share` constraint can be used to force groups of technologies to fulfill certain shares of supply or capacity.

For example, assuming a model containing a `csp` and a `cold_fusion` power generation technology, we could force at least 85% of power generation in the model to come from these two technologies with the following constraint definition in the model settings:

```yaml
model:
  group_share:
    csp, cold_fusion:
      carrier_prod_min:
        power: 0.85
```

Possible `group_share` constraints with carrier-specific settings are:

- `carrier_prod_min`
- `carrier_prod_max`
- `carrier_prod_equals`

Possible `group_share` constraints with carrier-independent settings are:

- `energy_cap_min`
- `energy_cap_max`
- `energy_cap_equals`

These can be implemented as, for example, to force at most 20% of `energy_cap` to come from the two listed technologies:

```yaml
model:
  group_share:
    csp, cold_fusion:
      energy_cap_max:
        energy_cap: 0.20
```

**Note:** The share given in the `carrier_prod` constraints refer to the use of generation from `supply` and `supply_plus` technologies only. The share given in the `energy_cap` constraints refers to the combined capacity from `supply`, `supply_plus`, `conversion`, and `conversion_plus` technologies.

See also:

The above examples are supplied as overrides in the *built-in national-scale example*’s scenarios. `yaml` (`cold_fusion` to define that tech, and `group_share_cold_fusion_prod` or `group_share_cold_fusion_cap` to apply the group share constraints).

### Removing techs, locations and links

By specifying `exists: false` in the model configuration, which can be done for example through overrides, model components can be removed for debugging or scenario analysis.

This works for:
Operational mode

In planning mode, constraints are given as upper and lower boundaries and the model decides on an optimal system configuration. In operational mode, all capacity constraints are fixed and the system is operated with a receding horizon control algorithm.

To specify a runnable operational model, capacities for all technologies at all locations must have been defined. This can be done by specifying `energy_cap_equals`. In the absence of `energy_cap_equals`, constraints given as `energy_cap_max` are assumed to be fixed in operational mode.

Operational mode runs a model with a receding horizon control algorithm. This requires two additional settings:

```
run:
  operation:
    horizon: 48 # hours
    window: 24 # hours
```

`horizon` specifies how far into the future the control algorithm optimises in each iteration. `window` specifies how many of the hours within `horizon` are actually used. In the above example, decisions on how to operate for each 24-hour window are made by optimising over 48-hour horizons (i.e., the second half of each optimisation run is discarded). For this reason, `horizon` must always be larger than `window`.

Generating scripts to run a model many times

Scenarios and overrides can be used to run a given model multiple times with slightly changed settings or constraints.

This functionality can be used together with the `calliope generate_runs` and `calliope generate_scenarios` command-line tools to generate scripts that run a model many times over in a fully automated way, for example, to explore the effect of different technology costs on model results.

`calliope generate_runs`, at a minimum, must be given the following arguments:

- the model configuration file to use
- the name of the script to create
- `--kind`: Currently, three options are available. `windows` creates a Windows batch (.bat) script that runs all models sequentially, `bash` creates an equivalent script to run on Linux or macOS, `bsub` creates a submission script for a LSF-based high-performance cluster, and `sbatch` creates a submission script for a SLURM-based high-performance cluster.
- `--scenarios`: A semicolon-separated list of scenarios (or overrides/combinations of overrides) to generate scripts for, for example, `scenario1;scenario2 or override1,override2a;override1,override2b`. Note that when not using manually defined scenario names, a comma is used to group overrides together into a single model – in the above example, `override1,override2a` would be applied to the first run and `override1,override2b` be applied to the second run.

A fully-formed command generating a Windows batch script to run a model four times with each of the scenarios “run1”, “run2”, “run3”, and “run4”:
calliope generate_runs model.yaml run_model.bat --kind=windows --scenarios "run1;run2;run3;run4"

Optional arguments are:

- `--cluster_threads`: specifies the number of threads to request on a HPC cluster
- `--cluster_mem`: specifies the memory to request on a HPC cluster
- `--cluster_time`: specifies the run time to request on a HPC cluster
- `--additional_args`: A text string of any additional arguments to pass directly through to calliope run in the generated scripts, for example, `--additional_args="--debug"`.
- `--debug`: Print additional debug information when running the run generation script.

An example generating a script to run on a `bsub`-type high-performance cluster, with additional arguments to specify the resources to request from the cluster:

calliope generate_runs model.yaml submit_runs.sh --kind=bsub --cluster_mem=1G --cluster_time=100 --cluster_threads=5 --scenarios "run1;run2;run3;run4"

Running this will create two files:

- `submit_runs.sh`: The cluster submission script to pass to `bsub` on the cluster.
- `submit_runs.array.sh`: The accompanying script defining the runs for the cluster to execute.

In all cases, results are saved into the same directory as the script, with filenames of the form `out_{run_number}_{scenario_name}.nc` (model results) and `plots_{run_number}_{scenario_name}.html` (HTML plots), where `{run_number}` is the run number and `{scenario_name}` is the name of the scenario (or the string defining the overrides applied). On a cluster, log files are saved to files with names starting with `log_` in the same directory.

Finally, the `calliope generate_scenarios` tool can be used to quickly generate a file with scenarios definition for inclusion in a model, if a large enough number of overrides exist to make it tedious to manually combine them into scenarios. Assuming that in `model.yaml` a range of overrides exist that specify a subset of time for the years 2000 through 2010, called “y2000” through “y2010”, and a set of cost-related overrides called “cost_low”, “cost_medium” and “cost_high”, the following command would generate scenarios with combinations of all years and cost overrides, calling them “run_1”, “run_2”, and so on, and saving them to `scenarios.yaml`:


**Imports in overrides**

When using overrides (see Scenarios and overrides), it is possible to have import statements within overrides for more flexibility. The following example illustrates this:

```yaml
overrides:
  some_override:
    techs:
      some_tech.constraints.energy_cap_max: 10
    import: [additional_definitions.yaml]
```

`additional_definitions.yaml`:
Binary and mixed-integer models

Calliope models are purely linear by default. However, several constraints can turn a model into a binary or mixed-integer model. Because solving problems with binary or integer variables takes considerably longer than solving purely linear models, it usually makes sense to carefully consider whether the research question really necessitates going beyond a purely linear model.

By applying a purchase cost to a technology, that technology will have a binary variable associated with it, describing whether or not it has been “purchased”.

By applying units.max, units.min, or units.equals to a technology, that technology will have an integer variable associated with it, describing how many of that technology have been “purchased”. If a purchase cost has been applied to this same technology, the purchasing cost will be applied per unit.

**Warning:** Integer and binary variables are a recent addition to Calliope and may not cover all edge cases as intended. Please raise an issue on GitHub if you see unexpected behavior.

See also:

*Tutorial 3: Mixed Integer Linear Programming*

Interfacing with the solver backend

On loading a model, there is no solver backend, only the input dataset. The backend is generated when a user calls `run()` on their model. Currently this will call back to Pyomo to build the model and send it off to the solver, given by the user in the run configuration `run.solver`. Once built, solved, and returned, the user has access to the results dataset `model.results` and interface functions with the backend `model.backend`.

You can use this interface to:

1. **Get the raw data on the inputs used in the optimisation.** By running `model.backend.get_input_params()` a user get an xarray Dataset which will look very similar to `model.inputs`, except that assumed default values will be included. You may also spot a bug, where a value in `model.inputs` is different to the value returned by this function.

2. **Update a parameter value.** If you are interested in updating a few values in the model, ou can run `model.backend.update_param()` . For example, to update your the energy efficiency of your ccgt technology in location `region1` from 0.5 to 0.1, you can run `model.backend.update_param('energy_eff', 'region1::ccgt', 0.1)`. This will not affect results at this stage, you’ll need to rerun the backend (point 4) to optimise with these new values.

3. **Activate / Deactivate a constraint or objective.** Constraints can be activated and deactivate such that they will or will not have an impact on the optimisation. All constraints are active by default, but you might like to
remove, for example, a capacity constraint if you don’t want there to be a capacity limit for any technologies. Similarly, if you had multiple objectives, you could deactivate one and activate another. The result would be to have a different objective when rerunning the backend.

**Note:** Currently Calliope does not allow you to build multiple objectives, you will need to understand Pyomo and add an additional objective yourself to make use of this functionality. The Pyomo ConcreteModel() object can be accessed at `model._backend_model`.

4. **Rerunning the backend.** If you have edited parameters or constraint activation, you will need to rerun the optimisation to propagate the effects. By calling `model.backend.rerun()`, the optimisation will run again, with the updated backend. This will not affect your model, but instead will return a dataset of the inputs/results associated with that specific rerun. It is up to you to store this dataset as you see fit. `model.results` will remain to be the initial run, and can only be overwritten by `model.run(force_rerun=True)`.

**Note:** By calling `model.run(force_rerun=True)` any updates you have made to the backend will be overwritten.

**See also:**

*Pyomo backend interface*

**Debugging failing runs**

A Calliope model provides a method to save a fully built and commented model to a single YAML file with `Model.save_commented_model_yaml(path)`. Comments in the resulting YAML file indicate where values were overridden.

Because this is Calliope’s internal representation of a model directly before the `model_data xarray.Dataset` is built, it can be useful for debugging possible issues in the model formulation, for example, undesired constraints that exist at specific locations because they were specified model-wide without having been superseded by location-specific settings.

Two configuration settings can further aid in debugging failing models:

- `model.subset_time` allows specifying a subset of timesteps to be used. This can be useful for debugging purposes as it can dramatically speed up model solution times. The timestep subset can be specified as `[startdate, enddate]`, e.g. `['2005-01-01', '2005-01-31']`, or as a single time period, such as 2005-01 to select January only. The subsets are processed before building the model and applying time resolution adjustments, so time resolution reduction functions will only see the reduced set of data.

- `run.save_logs` Off by default, if given, sets the directory into which to save logs and temporary files from the backend, to inspect solver logs and solver-generated model files. This also turns on symbolic solver labels in the Pyomo backend, so that all model components in the backend model are named according to the corresponding Calliope model components (by default, Pyomo uses short random names for all generated model components).

**See also:**

If using Calliope interactively in a Python session, we recommend reading up on the Python debugger and (if using Jupyter notebooks) making use of the `%debug magic`. 
Solver options

Gurobi

Refer to the Gurobi manual, which contains a list of parameters. Simply use the names given in the documentation (e.g. “NumericFocus” to set the numerical focus value). For example:

```
run:
  solver: gurobi
  solver_options:
    Threads: 3
    NumericFocus: 2
```

CPLEX

Refer to the CPLEX parameter list. Use the “Interactive” parameter names, replacing any spaces with underscores (for example, the memory reduction switch is called “emphasis memory”, and thus becomes “emphasis_memory”). For example:

```
run:
  solver: cplex
  solver_options:
    mipgap: 0.01
    mip_polishafter_absmipgap: 0.1
    emphasis_mip: 1
    mip_cuts: 2
    mip_cuts_cliques: 3
```

1.7.2 Built-in example models

This section gives a listing of all the YAML configuration files included in the built-in example models. Refer to the tutorials section for a brief overview of how these parts together provide a working model.

The example models are accessible in the calliope.examples module. To create an instance of an example model, call its constructor function, e.g.:

```
urban_model = calliope.examples.urban_scale()
```

The available example models and their constructor functions are:

calliope.examples.national_scale(*args, **kwargs)
Returns the built-in national-scale example model.

calliope.examples.time_clustering(*args, **kwargs)
Returns the built-in national-scale example model with time clustering.

calliope.examples.time_resampling(*args, **kwargs)
Returns the built-in national-scale example model with time resampling.

calliope.examples.urban_scale(*args, **kwargs)
Returns the built-in urban-scale example model.

calliope.examples.milp(*args, **kwargs)
Returns the built-in urban-scale example model with MILP constraints enabled.
calliope.examples.**operate**(*args, **kwargs*)

Returns the built-in urban-scale example model in operate mode.

calliope.examples.**time_masking**(*args, **kwargs*)

Returns the built-in urban-scale example model with time masking.

**National-scale example**

Available as `calliope.examples.national_scale`.

**Model settings**

The layout of the model directory is as follows (+ denotes directories, - files):

- model.yaml
- scenarios.yaml
+ timeseries_data
  - csp_resource.csv
  - demand-1.csv
  - demand-2.csv
+ model_config
  - locations.yaml
  - techs.yaml

**model.yaml**:

```yaml
import:  # Import other files from paths relative to this file, or absolute paths
  - 'model_config/techs.yaml'  # This file specifies the model's technologies
  - 'model_config/locations.yaml'  # This file specifies the model's locations
  - 'scenarios.yaml'  # Scenario and override group definitions

# Model configuration: all settings that affect the built model
model:
  name: National-scale example model

  # What version of Calliope this model is intended for
  calliope_version: 0.6.3

  # Time series data path - can either be a path relative to this file, or an
  # absolute path
  timeseries_data_path: 'timeseries_data'

  subset_time: ['2005-01-01', '2005-01-05']  # Subset of timesteps

# Run configuration: all settings that affect how the built model is run
run:
  solver: glpk

  ensure_feasibility: true  # Switches on the "unmet demand" constraint

  bigM: 1e6  # Sets the scale of unmet demand, which cannot be too high, otherwise
  # the optimisation will not converge

  zero_threshold: 1e-10  # Any value coming out of the backend that is smaller than
  # this (due to floating point errors, probably) will be set to zero
```

(continues on next page)
mode: plan  # Choices: plan, operate

scenarios.yaml:

## # Scenarios are optional, named combinations of overrides
## scenarios:
cold_fusion_with_production_share: 'cold_fusion,group_share_cold_fusion_prod'
cold_fusion_with_capacity_share: 'cold_fusion,group_share_cold_fusion_cap'

## # Overrides are the building blocks from which scenarios can be defined
## overrides:
  profiling:
    model.name: 'National-scale example model (profiling run)'
    run.solver: glpk

time_resampling:
    model.name: 'National-scale example model with time resampling'
    model.subset_time: '2005-01'
    # Resample time resolution to 6-hourly
    model.time: {function: resample, function_options: {resolution: '6H'}}

time_clustering:
    model.random_seed: 23
    model.name: 'National-scale example model with time clustering'
    model.subset_time: null  # No time subsetting
    # Cluster timesteps using k-means
    model.time: {function: apply_clustering, function_options: {clustering_func: 'kmeans', how: 'closest', k: 10}}

operate:
  run.mode: operate
  run.operation:
    window: 12
    horizon: 24
  model.subset_time: ['2005-01-01', '2005-01-10']

locations:
  region1.techs.ccgts.constraints.energy_cap_equals: 30000
  region2.techs.battery.constraints.energy_cap_equals: 1000
  region2.techs.battery.constraints.storage_cap_equals: 5240

  region1-1.techs.csp.constraints.energy_cap_equals: 10000
  region1-1.techs.csp.constraints.storage_cap_equals: 244301
  region1-1.techs.csp.constraints.resource_area_equals: 130385

  region1-2.techs.csp.constraints.energy_cap_equals: 0
  region1-2.techs.csp.constraints.storage_cap_equals: 0
  region1-2.techs.csp.constraints.resource_area_equals: 0

  region1-3.techs.csp.constraints.energy_cap_equals: 2534
region1-3.techs.csp.constraints.storage_cap_equals: 25301
region1-3.techs.csp.constraints.resource_area_equals: 8487

links:
region1,region2.techs.ac_transmission.constraints.energy_cap_equals: 3231
region1,region1-1.techs.free_transmission.constraints.energy_cap_equals: 9000
region1,region1-2.techs.free_transmission.constraints.energy_cap_equals: 0
region1,region1-3.techs.free_transmission.constraints.energy_cap_equals: 2281

check_feasibility:
  run:
    ensure_feasibility: False
    objective: 'check_feasibility'
  model:
    subset_time: '2005-01-04'

reserve_margin:
  model:
    # Model-wide settings for the system-wide reserve margin
    # Even setting a reserve margin of zero activates the constraint,
    # forcing enough installed capacity to cover demand in
    # the maximum demand timestep
    reserve_margin:
      power: 0.10  # 10% reserve margin for power

##
# Overrides to demonstrate the run generator ("calliope generate_runs")
#
run1:
run2:
  model.subset_time: ['2005-02-01', '2005-02-31']
run3:
  locations.region1.techs.ccgt.constraints.energy_cap_max: 0  # Disallow CCGT
run4:
  subset_time: ['2005-02-01', '2005-02-31']
  locations.region1.techs.ccgt.constraints.energy_cap_max: 0  # Disallow CCGT

##
# Overrides to demonstrate the group_share constraints
#
cold_fusion:  # Defines a hypothetical cold fusion tech to use in group_share
techs:
  cold_fusion:
    essentials:
      name: 'Cold fusion'
      color: '#233B39'
      parent: supply
      carrier_out: power
    constraints:
      energy_cap_max: 10000
      lifetime: 50

(continues on next page)
costs:
    monetary:
        interest_rate: 0.20
        energy_cap: 100
    locations.region1.techs.cold_fusion: null
    locations.region2.techs.cold_fusion: null

    group_share_cold_fusion_prod:
        model:
            group_share:
                # At least 85% of power supply must come from CSP and cold fusion
                csp,cold_fusion:
                    carrier_prod_min:
                        power: 0.85

    group_share_cold_fusion_cap:
        model:
            group_share:
                # At most 20% of total energy_cap can come from CSP and cold fusion
                csp,cold_fusion:
                    energy_cap_max: 0.20

    locations:
        region1:
            techs:
                ccgt:
                    constraints:
                        energy_cap_max: 100000  # Increased to keep model feasible

minimize_emissions_costs:
    run:
        objective_options:
            cost_class: emissions
techs:
    ccgt:
        costs:
            emissions:
                om_prod: 100  # kgCO2/kWh
csp:
        costs:
            emissions:
                om_prod: 10  # kgCO2/kWh

maximize_utility_costs:
    run:
        objective_options:
            cost_class: utility
            sense: maximize
techs:
    ccgt:
        costs:
            utility:
                om_prod: 10  # arbitrary utility value
csp:
        costs:
            utility:
om_prod: 100  # arbitrary utility value

techs.yaml:

## TECHNOLOGY DEFINITIONS
##
# Note: '-start' and '-end' is used in tutorial documentation only
# ccgt-start
ccgt:
  essentials:
    name: 'Combined cycle gas turbine'
    color: '#E37A72'
    parent: supply
    carrier_out: power
  constraints:
    resource: inf
    energy_eff: 0.5
    energy_cap_max: 40000  # kW
    energy_cap_max_systemwide: 100000  # kW
    energy_ramping: 0.8
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap: 750  # USD per kW
      om_con: 0.02  # USD per kWh
# ccgt-end

# csp-start

csp:
  essentials:
    name: 'Concentrating solar power'
    color: '#F9CF22'
    parent: supply_plus
    carrier_out: power
  constraints:
    storage_cap_max: 614033
    charge_rate: 1
    storage_loss: 0.002
    resource: file=csp_resource.csv
    resource_unit: energy_per_area
    energy_eff: 0.4
    parasitic_eff: 0.9
    resource_area_max: inf
    energy_cap_max: 10000
    lifetime: 25
  costs:
    monetary:
interest_rate: 0.10
storage_cap: 50
resource_area: 200
resource_cap: 200
energy_cap: 1000
on_prod: 0.002

# csp-end

## Storage

# battery-start
battery:
  essentials:
    name: 'Battery storage'
    color: '#3B61E3'
    parent: storage
    carrier: power
  constraints:
    energy_cap_max: 1000  # kW
    storage_cap_max: inf
    charge_rate: 4
    energy_eff: 0.95  # 0.95 * 0.95 = 0.9025 round trip efficiency
    storage_loss: 0  # No loss over time assumed
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      storage_cap: 200  # USD per kWh storage capacity

# battery-end

## Demand

## demand-start
demand_power:
  essentials:
    name: 'Power demand'
    color: '#072486'
    parent: demand
    carrier: power

# demand-end

## Transmission

## transmission-start
ac_transmission:
  essentials:
    name: 'AC power transmission'
    color: '#8465A9'
    parent: transmission
    carrier: power
  constraints:
    energy_eff: 0.85
    lifetime: 25

(continues on next page)
Calliope Documentation, Release 0.6.3

---

```
costs:
  monetary:
    interest_rate: 0.10
    energy_cap: 200
    om_prod: 0.002

free_transmission:
  essentials:
    name: 'Local power transmission'
    color: '#6783E3'
    parent: transmission
    carrier: power
    constraints:
      energy_cap_max: inf
      energy_eff: 1.0
    costs:
      monetary:
        om_prod: 0
      # transmission-end

locations.yaml:

# LOCATIONS

locations:
  # region-1-start
  region1:
    coordinates: {lat: 40, lon: -2}
    techs:
      demand_power:
        constraints:
          resource: file=demand-1.csv:demand
        ccgt:
          constraints:
            energy_cap_max: 30000 # increased to ensure no unmet_demand in first timestep
  # region-1-end
  # other-locs-start
  region2:
    coordinates: {lat: 40, lon: -8}
    techs:
      demand_power:
        constraints:
          resource: file=demand-2.csv:demand
      battery:
        region1-1.coordinates: {lat: 41, lon: -2}
        region1-2.coordinates: {lat: 39, lon: -1}
        region1-3.coordinates: {lat: 39, lon: -2}
        region1-1, region1-2, region1-3:
          techs:
            csp:
            # other-locs-end
```

---

(continued on next page)
## TRANSMISSION CAPACITIES

links:

```yaml
# links-start
region1,region2:
  techs:
    ac_transmission:
      constraints:
        energy_cap_max: 10000
region1,region1-1:
  techs:
    free_transmission:
region1,region1-2:
  techs:
    free_transmission:
region1,region1-3:
  techs:
    free_transmission:
# links-end
```

### Urban-scale example

Available as `calliope.examples.urban_scale`.

### Model settings

model.yaml:

```yaml
import: # Import other files from paths relative to this file, or absolute paths
  - 'model_config/techs.yaml'
  - 'model_config/locations.yaml'
  - 'scenarios.yaml'

model:
  name: Urban-scale example model

  calliope_version: 0.6.3

  timeseries_data_path: 'timeseries_data'

  subset_time: ['2005-07-01', '2005-07-02'] # Subset of timesteps

run:
  mode: plan # Choices: plan, operate
  solver: glpk

  ensure_feasibility: true # Switching on unmet demand
```

(continues on next page)
bigM: 1e6  # setting the scale of unmet demand, which cannot be too high, otherwise the optimisation will not converge

scenarios.yaml:

```yaml
##
# Overrides for different example model configurations
##

overrides:
  milp:
    model.name: 'Urban-scale example model with MILP'
    techs:
      # chp-start
      chp:
        constraints:
          units_max: 4
          energy_cap_per_unit: 300
          energy_cap_min_use: 0.2
        costs:
          monetary:
            energy_cap: 700
            purchase: 40000
      # chp-end
      # boiler-start
      boiler:
        costs:
          monetary:
            energy_cap: 35
            purchase: 2000
      # boiler-end

mapbox_ready:
  locations:
    X1.coordinates: {lat: 51.4596158, lon: -0.1613446}
    X2.coordinates: {lat: 51.4652373, lon: -0.1141548}
    X3.coordinates: {lat: 51.4287016, lon: -0.1310635}
    N1.coordinates: {lat: 51.4450766, lon: -0.1247183}
  links:
    X1,X2.techs.power_lines.distance: 10
    X1,X3.techs.power_lines.distance: 5
    X1,N1.techs.heat_pipes.distance: 3
    N1,X2.techs.heat_pipes.distance: 3
    N1,X3.techs.heat_pipes.distance: 4

operate:
  run.mode: operate
  run.operation:
    window: 24
    horizon: 48
  model.subset_time: ['2005-07-01', '2005-07-10']
  locations:
    X1:
      techs:
        chp.constraints.energy_cap_max: 300
        pv.constraints.energy_cap_max: 0
```

(continues on next page)
supply_grid_power.constraints.energy_cap_max: 40
supply_gas.constraints.energy_cap_max: 700

X2:
technologies:
  - boiler.constraints.energy_cap_max: 200
  - pv.constraints.energy_cap_max: 70
  - supply_gas.constraints.energy_cap_max: 250

X3:
technologies:
  - boiler.constraints.energy_cap_max: 0
  - pv.constraints.energy_cap_max: 50
  - supply_gas.constraints.energy_cap_max: 0

links:
  - X1,X2.technologies.power_lines.constraints.energy_cap_max: 300
  - X1,X3.technologies.power_lines.constraints.energy_cap_max: 60
  - X1,N1.technologies.heat_pipes.constraints.energy_cap_max: 300
  - N1,X2.technologies.heat_pipes.constraints.energy_cap_max: 250
  - N1,X3.technologies.heat_pipes.constraints.energy_cap_max: 320

time_masking:
model.name: 'Urban-scale example model with time masking'
model.subset_time: '2005-01'
# Resample time resolution to 6-hourly
model.time:
masks:
  - function: extreme_diff, options: {tech0: demand_heat, tech1: demand_electricity, how: max, n: 2}
    function: resample
    function_options: {resolution: 6H}

technologies.yml:

## TECHNOLOGY DEFINITIONS
##
# Note: '-start' and '-end' is used in tutorial documentation only

# supply_power_plus-start
technologies:
  supply_power_plus:
    essentials:
      parent: supply_plus
      carrier: electricity
# supply_power_plus-end

# supply-start

technologies:

##-GRID SUPPLY-##
# supply-start
supply_grid_power:
  essentials:
    name: 'National grid import'
    color: '#C5ABE3'
parent: supply
carrier: electricity
constraints:
    resource: inf
    energy_cap_max: 2000
    lifetime: 25
costs:
    monetary:
        interest_rate: 0.10
        energy_cap: 15
        om_con: 0.1 # 10p/kWh electricity price #ppt

supply_gas:
    essentials:
        name: 'Natural gas import'
        color: '#C98AAD'
        parent: supply
carrier: gas
constraints:
    resource: inf
    energy_cap_max: 2000
    lifetime: 25
costs:
    monetary:
        interest_rate: 0.10
        energy_cap: 1
        om_con: 0.025 # 2.5p/kWh gas price #ppt
    # supply-end

##-Renewables-##

### pv-start ###

# pv:

pv:
    essentials:
        name: 'Solar photovoltaic power'
        color: '#F9D956'
        parent: supply_power_plus
constraints:
    export_carrier: electricity
    resource: file=pv_resource.csv:per_area
    # Already accounts for panel efficiency - kWh/m2. Source: Renewables.ninja Solar PV Power - Version: 1.1 - License: https://creativecommons.org/licenses/by-nc/4.0/ - Reference: https://doi.org/10.1016/j.energy.2016.08.060
    resource_unit: energy_per_area
    parasitic_eff: 0.85 # inverter losses
    energy_cap_max: 250
    resource_area_max: 1500
    force_resource: true
    resource_area_per_energy_cap: 7 # 7m2 of panels needed to fit 1kWp of panels
    lifetime: 25
    costs:
        monetary:
            interest_rate: 0.10
            energy_cap: 1350
    # pv-end

# Conversion

(continues on next page)
boiler:
  essentials:
    name: 'Natural gas boiler'
    color: '#8E2999'
    parent: conversion
    carrier_out: heat
    carrier_in: gas
  constraints:
    energy_cap_max: 600
    energy_eff: 0.85
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      om_con: 0.004  # .4p/kWh
boiler:
# Conversion_plus
# chp:
chp:
  essentials:
    name: 'Combined heat and power'
    color: '#E4AB97'
    parent: conversion_plus
    primary_carrier_out: electricity
    carrier_in: gas
    carrier_out: electricity
    carrier_out_2: heat
  constraints:
    export_carrier: electricity
    energy_cap_max: 1500
    energy_eff: 0.405
    carrier_ratios.carrier_out_2.heat: 0.8
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap: 750
      om_prod: 0.004  # .4p/kWh for 4500 operating hours/year
      export: file=export_power.csv
chp:
## DEMAND ##
demand_electricity:
  essentials:
    name: 'Electrical demand'
    color: '#072486'
    parent: demand
    carrier: electricity
demand_heat:
  essentials:
    name: 'Heat demand'
    color: '#660507'
    parent: demand
carrier: heat
# demand-end

## DISTRIBUTION ##
# transmission-start
power_lines:
  essentials:
    name: 'Electrical power distribution'
    color: '#6783E3'
    parent: transmission
    carrier: electricity
  constraints:
    energy_cap_max: 2000
    energy_eff: 0.98
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap_per_distance: 0.01

heat_pipes:
  essentials:
    name: 'District heat distribution'
    color: '#823739'
    parent: transmission
    carrier: heat
  constraints:
    energy_cap_max: 2000
    energy_eff_per_distance: 0.975
    lifetime: 25
  costs:
    monetary:
      interest_rate: 0.10
      energy_cap_per_distance: 0.3
# transmission-end

locations.yaml:
locations:
  # X1-start
  X1:
    techs:
      chp:
      pv:
        supply_grid_power:
          costs.monetary.energy_cap: 100
          # cost of transformers
        supply_gas:
        demand_electricity:
          constraints.resource: file=demand_power.csv
        demand_heat:
          constraints.resource: file=demand_heat.csv
        available_area: 500
        coordinates: {x: 2, y: 7}
  # X1-end
  # other-locs-start
  X2:
    techs:
boiler:
  costs.monetary.energy_cap: 43.1  # different boiler costs
pv:
  costs.monetary:
    om_prod: -0.0203  # revenue for just producing electricity
    export: -0.0491  # FIT return for PV export
supply_gas:
demand_electricity:
  constraints.resource: file=demand_power.csv
demand_heat:
  constraints.resource: file=demand_heat.csv
available_area: 1300
coordinates: {x: 8, y: 7}

X3:
  techs:
    boiler:
      costs.monetary.energy_cap: 78  # different boiler costs
    pv:
      constraints:
        energy_cap_max: 50  # changing tariff structure below 50kW
      costs.monetary:
        om_annual: -80.5  # reimbursement per kWp from FIT
  supply_gas:
demand_electricity:
  demand_heat:
  available_area: 900
coordinates: {x: 5, y: 3}

# other-locs-end
# N1-start
 N1:  # location for branching heat transmission network
  coordinates: {x: 5, y: 7}
# N1-end

links:
  # links-start
  X1,X2:
    techs:
      power_lines:
        distance: 10
  X1,X3:
    techs:
      power_lines:
  X1,N1:
    techs:
      heat_pipes:
  N1,X2:
    techs:
      heat_pipes:
  N1,X3:
    techs:
      heat_pipes:
  # links-end
1.7.3 Listing of configuration options

Configuration layout

There must always be at least one model configuration YAML file, probably called `model.yaml` or similar. This file can import any number of additional files.

This file or this set of files must specify the following top-level configuration keys:

- **name**: the name of the model
- **model**: model settings
- **run**: run settings
- **techs**: technology definitions
- (optionally) **tech_groups**: tech group definitions
- **locations**: location definitions
- (optionally) **links**: transmission link definitions

**Note**: Model settings (model) affect how the model and its data are built by Calliope, while run settings (run) only take effect once a built model is run (e.g. interactively via `model.run()`). This means that run settings, unlike model settings, can be updated after a model is built and before it is run, by modifying attributes in the built model dataset.

### List of model settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>calliope_version</td>
<td></td>
<td>Calliope framework version this model is intended for</td>
</tr>
<tr>
<td>group_share</td>
<td>{}</td>
<td>Optional settings for the group_share constraint</td>
</tr>
<tr>
<td>name</td>
<td></td>
<td>Model name</td>
</tr>
<tr>
<td>random_seed</td>
<td></td>
<td>Seed for random number generator used during clustering</td>
</tr>
<tr>
<td>reserve_margin</td>
<td>{}</td>
<td>Per-carrier system-wide reserve margins</td>
</tr>
<tr>
<td>subset_time</td>
<td></td>
<td>Subset of timesteps as a two-element list giving the range, e.g. ['2005-01-01', '2005-01-05'], or a single string, e.g. '2005-01'</td>
</tr>
<tr>
<td>time</td>
<td>{}</td>
<td>Optional settings to adjust time resolution, see <code>Time resolution adjustment</code> for the available options</td>
</tr>
<tr>
<td>timeseries_data_path</td>
<td></td>
<td>Path to time series data</td>
</tr>
<tr>
<td>timeseries_data</td>
<td></td>
<td>Dict of dataframes with time series data (when passing in dicts rather than YAML files to Model constructor)</td>
</tr>
<tr>
<td>timeseries_dateformat</td>
<td><code>%Y-%m-%d %H:%M:%S</code></td>
<td>Timestamp format of all time series data when read from file</td>
</tr>
</tbody>
</table>
List of run settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend</td>
<td>pyomo</td>
<td>Backend to use to build and solve the model. As of v0.6.0, only pyomo is available</td>
</tr>
<tr>
<td>bigM</td>
<td>1000000000.0</td>
<td>Used for unmet demand, but should be of a similar order of magnitude as the largest cost that the model could achieve. Too high and the model will not converge</td>
</tr>
<tr>
<td>cyclic_storage</td>
<td>True</td>
<td>If true, storage in the last timestep of the timeseries is considered to be the ‘previous timestep’ in the first timestep of the timeseries</td>
</tr>
<tr>
<td>ensure_feasibility</td>
<td>False</td>
<td>If true, unmet_demand will be a decision variable, to account for an ability to meet demand with the available supply. If False and a mismatch occurs, the optimisation will fail due to infeasibility</td>
</tr>
<tr>
<td>mode</td>
<td>plan</td>
<td>Which mode to run the model in: ‘plan’ or ‘operation’</td>
</tr>
<tr>
<td>objective_options</td>
<td>{}</td>
<td>Arguments to pass to objective function. If cost-based objective function in use, should include ‘cost_class’ and ‘sense’ (maximize/minimize)</td>
</tr>
<tr>
<td>objective</td>
<td>minmax_cost_optimization</td>
<td>Name of internal objective function to use, currently only min/max cost-based optimisation is available</td>
</tr>
<tr>
<td>operation</td>
<td>{}</td>
<td>Settings for operational mode</td>
</tr>
<tr>
<td>save_logs</td>
<td></td>
<td>Directory into which to save logs and temporary files. Also turns on symbolic solver labels in the Pyomo backend</td>
</tr>
<tr>
<td>solver_io</td>
<td></td>
<td>What method the Pyomo backend should use to communicate with the solver</td>
</tr>
<tr>
<td>solver_options</td>
<td>A list of options, which are passed on to the chosen solver, and are therefore solver-dependent</td>
<td></td>
</tr>
<tr>
<td>solver</td>
<td>glpk</td>
<td>Which solver to use</td>
</tr>
<tr>
<td>zero_threshold</td>
<td>1e-10</td>
<td>Any value coming out of the backend that is smaller than this threshold (due to floating point errors, probably) will be set to zero</td>
</tr>
</tbody>
</table>

List of possible constraints

The following table lists all available technology constraint settings and their default values. All of these can be set by tech_identifier.constraints.constraint_name, e.g. nuclear.constraints.e_cap.max.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Name</th>
<th>Unit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>carrier_ratios</td>
<td>{}</td>
<td>Carrier ratios</td>
<td>fraction</td>
<td>Ratio of summed output of carriers in ['out_2', 'out_3'] / ['in_2', 'in_3'] to the summed output of carriers in 'out' / 'in'. given in a nested dictionary.</td>
</tr>
<tr>
<td>charge_rate</td>
<td>False</td>
<td>Charge rate</td>
<td>hour⁻¹</td>
<td>ratio of maximum charge/discharge (kW) for a given maximum storage capacity (kWh).</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Name</th>
<th>Unit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy_cap_equals</td>
<td>False</td>
<td>Specific installed energy capacity</td>
<td>kW</td>
<td>fixes maximum/minimum if decision variables carrier_prod/carrier_con and overrides _max and _min constraints.</td>
</tr>
<tr>
<td>energy_cap_equals_systemwide</td>
<td>False</td>
<td>System-wide specific installed energy capacity</td>
<td>kW</td>
<td>fixes the sum to a maximum/minimum, for a particular technology, of the decision variables carrier_prod/carrier_con over all locations.</td>
</tr>
<tr>
<td>energy_cap_max</td>
<td>inf</td>
<td>Maximum installed energy capacity</td>
<td>kW</td>
<td>Limits decision variables carrier_prod/carrier_con to a maximum/minimum.</td>
</tr>
<tr>
<td>energy_cap_max_systemwide</td>
<td>inf</td>
<td>System-wide maximum installed energy capacity</td>
<td>kW</td>
<td>Limits the sum to a maximum/minimum, for a particular technology, of the decision variables carrier_prod/carrier_con over all locations.</td>
</tr>
<tr>
<td>energy_cap_min</td>
<td>0</td>
<td>Minimum installed energy capacity</td>
<td>kW</td>
<td>Limits decision variables carrier_prod/carrier_con to a minimum/maximum.</td>
</tr>
<tr>
<td>energy_cap_min_use</td>
<td>False</td>
<td>Minimum carrier production</td>
<td>fraction</td>
<td>Set to a value between 0 and 1 to force minimum carrier production as a fraction of the technology maximum energy capacity. If non-zero and technology is not defined by units, this will force the technology to operate above its minimum value at every timestep.</td>
</tr>
<tr>
<td>energy_cap_per_unit</td>
<td>False</td>
<td>Energy capacity per purchased unit</td>
<td>kW/unit</td>
<td>Set the capacity of each integer unit of a technology purchased.</td>
</tr>
<tr>
<td>energy_cap_scale</td>
<td>1.0</td>
<td>Energy capacity scale</td>
<td>float</td>
<td>Scale all energy_cap_min/max/equals/total_max/total_equals constraints by this value.</td>
</tr>
<tr>
<td>energy_con</td>
<td>False</td>
<td>Energy consumption</td>
<td>boolean</td>
<td>Allow this technology to consume energy from the carrier (static boolean, or from file as timeseries).</td>
</tr>
<tr>
<td>energy_eff</td>
<td>1.0</td>
<td>Energy efficiency</td>
<td>fraction</td>
<td>conversion efficiency (static, or from file as timeseries), from resource/storage/carrier_in (tech dependent) to carrier_out.</td>
</tr>
<tr>
<td>energy_eff_per_distance</td>
<td>1.0</td>
<td>Energy efficiency per distance</td>
<td>distance</td>
<td>Set as value between 1 (no loss) and 0 (all energy lost).</td>
</tr>
<tr>
<td>energyProd</td>
<td>False</td>
<td>Energy production</td>
<td>boolean</td>
<td>Allow this technology to supply energy to the carrier (static boolean, or from file as timeseries).</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Name</th>
<th>Unit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy_ramping</td>
<td>False</td>
<td>Ramping rate</td>
<td>fraction / hour</td>
<td>Set to false to disable ramping constraints, otherwise limit maximum carrier production to a fraction of maximum capacity, which increases by that fraction at each timestep.</td>
</tr>
<tr>
<td>export_cap</td>
<td>inf</td>
<td>Export capacity</td>
<td>kW</td>
<td>Maximum allowed export of produced energy carrier for a technology.</td>
</tr>
<tr>
<td>export_carrier</td>
<td></td>
<td>Export carrier</td>
<td>N/A</td>
<td>Name of carrier to be exported. Must be an output carrier of the technology.</td>
</tr>
<tr>
<td>force_resource</td>
<td>False</td>
<td>Force resource</td>
<td>boolean</td>
<td>Forces this technology to use all available resource, rather than making it a maximum upper boundary (for production) or minimum lower boundary (for consumption). Static boolean, or from file as timeseries.</td>
</tr>
<tr>
<td>lifetime</td>
<td></td>
<td>Technology lifetime</td>
<td>years</td>
<td>Must be defined if fixed capital costs are defined. A reasonable value for many technologies is around 20-25 years.</td>
</tr>
<tr>
<td>parasitic_eff</td>
<td>1.0</td>
<td>Plant parasitic efficiency</td>
<td>fraction</td>
<td>Additional losses as energy gets transferred from the plant to the carrier (static, or from file as timeseries), e.g. due to plant parasitic consumption</td>
</tr>
<tr>
<td>resource</td>
<td>0</td>
<td>Available resource</td>
<td>kWh/m²</td>
<td>kWh/m²</td>
</tr>
<tr>
<td>resource_area_equals</td>
<td>False</td>
<td>Specific installed collector area</td>
<td>m²</td>
<td></td>
</tr>
<tr>
<td>resource_area_max</td>
<td>False</td>
<td>Maximum installed collector area</td>
<td>m²</td>
<td>Set to false by default in order to disable this constraint.</td>
</tr>
<tr>
<td>resource_area_min</td>
<td>0</td>
<td>Minimum installed collector area</td>
<td>m²</td>
<td></td>
</tr>
<tr>
<td>resource_area_per_energy_cap</td>
<td>False</td>
<td>Energy capacity per unit collector area</td>
<td>boolean</td>
<td>If set, forces resource_area to follow energy_cap with the given numerical ratio (e.g. setting to 1.5 means that resource_area == 1.5 * energy_cap).</td>
</tr>
<tr>
<td>resource_cap_equals</td>
<td>False</td>
<td>Specific installed resource consumption capacity</td>
<td>kW</td>
<td>overrides _max and _min constraints.</td>
</tr>
<tr>
<td>resource_cap_equals_energy_cap</td>
<td>False</td>
<td>Resource capacity equals energy capacity</td>
<td>boolean</td>
<td>If true, resource_cap is forced to equal energy_cap.</td>
</tr>
<tr>
<td>resource_cap_max</td>
<td>inf</td>
<td>Maximum installed resource consumption capacity</td>
<td>kW</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Name</th>
<th>Unit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource_cap_min</td>
<td>0</td>
<td>Minimum installed resource consumption</td>
<td>kW</td>
<td>Efficiency (static, or from file as timeseries) in capturing resource before it reaches storage (if storage is present) or conversion to carrier.</td>
</tr>
<tr>
<td>resource_eff</td>
<td>1.0</td>
<td>Resource efficiency</td>
<td>fraction</td>
<td>Set to a value between 0 and 1 to force minimum resource consumption for production technologies</td>
</tr>
<tr>
<td>resource_min_use</td>
<td>False</td>
<td>Minimum resource consumption</td>
<td>fraction</td>
<td>Scale resource (either static value or all values in timeseries) by this value</td>
</tr>
<tr>
<td>resource_scale</td>
<td>1.0</td>
<td>Resource scale</td>
<td>fraction</td>
<td>Sets the unit of resource to either energy (i.e., kWh), energy_per_area (i.e., kWh/m²), or energy_per_cap (i.e., kWh/kW). energy_per_area uses the resource_area decision variable to scale the available resource while energy_per_cap uses the energy_cap decision variable.</td>
</tr>
<tr>
<td>resource_unit</td>
<td>energy</td>
<td>Resource unit</td>
<td>N/A</td>
<td>Sets the specific installed energy capacity</td>
</tr>
<tr>
<td>storage_cap_equals</td>
<td>False</td>
<td>Specific storage capacity</td>
<td>kWh</td>
<td>If not defined, energy_cap_equals * charge_rate will be used as the capacity and overrides _max and _min constraints.</td>
</tr>
<tr>
<td>storage_cap_max</td>
<td>inf</td>
<td>Maximum storage capacity</td>
<td>kWh</td>
<td>If not defined, energy_cap_max * charge_rate will be used as the capacity.</td>
</tr>
<tr>
<td>storage_cap_min</td>
<td>0</td>
<td>Minimum storage capacity</td>
<td>kWh</td>
<td></td>
</tr>
<tr>
<td>storage_cap_per_unit</td>
<td>False</td>
<td>Storage capacity per purchased unit</td>
<td>kWh/unit</td>
<td>Set the storage capacity of each integer unit of a technology purchased.</td>
</tr>
<tr>
<td>storage_initial</td>
<td>0</td>
<td>Initial storage level</td>
<td>kWh</td>
<td>Set stored energy in device at the first timestep</td>
</tr>
<tr>
<td>storage_loss</td>
<td>0</td>
<td>Storage loss rate</td>
<td>hour⁻¹</td>
<td>rate of storage loss per hour (static, or from file as timeseries), used to calculate lost stored energy as (1 - storage_loss)⁻¹ * storage_cap * hours_per_timestep</td>
</tr>
<tr>
<td>units_equals</td>
<td>False</td>
<td>Specific number of purchased units</td>
<td>integer</td>
<td>Turns the model from LP to MILP.</td>
</tr>
<tr>
<td>units_equals_systemwide</td>
<td>False</td>
<td>System-wide specific installed energy capacity</td>
<td>kW</td>
<td>fixes the sum to a specific value, for a particular technology, of the decision variables carrier_prod/carrier_con over all locations.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Name</th>
<th>Unit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>units_max</td>
<td>False</td>
<td>Maximum number of purchased units</td>
<td>integer</td>
<td>Turns the model from LP to MILP.</td>
</tr>
<tr>
<td>units_max_systemwide</td>
<td>inf</td>
<td>System-wide maximum installed energy capacity</td>
<td>kW</td>
<td>Limits the sum to a maximum/minimum, for a particular technology, of the decision variables carrier_prod/carrier_con over all locations.</td>
</tr>
<tr>
<td>units_min</td>
<td>False</td>
<td>Minimum number of purchased units</td>
<td>integer</td>
<td>Turns the model from LP to MILP.</td>
</tr>
</tbody>
</table>

List of possible costs

These are all the available costs, which are set to 0 by default for every defined cost class. Costs are set by tech_identifier.costs.cost_class.cost_name, e.g. nuclear.costs.monetary.e_cap.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Name</th>
<th>Unit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy_cap</td>
<td>0</td>
<td>Cost of energy capacity</td>
<td>kW/gross -1</td>
<td></td>
</tr>
<tr>
<td>energy_cap_per_distance</td>
<td>0</td>
<td>Cost of energy capacity, per unit distance</td>
<td>kW/gross -1 / distance</td>
<td>Applied to transmission links only</td>
</tr>
<tr>
<td>export</td>
<td>0</td>
<td>Carrier export cost</td>
<td>kWh -1</td>
<td>Usually used in the negative sense, as a subsidy.</td>
</tr>
<tr>
<td>interest_rate</td>
<td>0</td>
<td>Interest rate</td>
<td>fraction</td>
<td>Used when computing levelized costs</td>
</tr>
<tr>
<td>om_annual</td>
<td>0</td>
<td>Yearly O&amp;M costs</td>
<td>kW/energy_cap -1</td>
<td></td>
</tr>
<tr>
<td>om_annual_investment_fraction</td>
<td>0</td>
<td>Fractional yearly O&amp;M costs</td>
<td>fraction / total investment</td>
<td>Applied to carrier consumption of a technology</td>
</tr>
<tr>
<td>om_con</td>
<td>0</td>
<td>Carrier consumption cost</td>
<td>kWh -1</td>
<td>Applied to carrier consumption of a technology</td>
</tr>
<tr>
<td>om_prod</td>
<td>0</td>
<td>Carrier production cost</td>
<td>kWh -1</td>
<td>Applied to carrier production of a technology</td>
</tr>
<tr>
<td>purchase</td>
<td>0</td>
<td>Purchase cost</td>
<td>unit -1</td>
<td>Triggers a binary variable for that technology to say that it has been purchased or is applied to integer variable units</td>
</tr>
<tr>
<td>resource_area</td>
<td>0</td>
<td>Cost of resource collector area</td>
<td>m²</td>
<td></td>
</tr>
<tr>
<td>resource_cap</td>
<td>0</td>
<td>Cost of resource consumption capacity</td>
<td>kW -1</td>
<td></td>
</tr>
<tr>
<td>storage_cap</td>
<td>0</td>
<td>Cost of storage capacity</td>
<td>kWh -1</td>
<td></td>
</tr>
</tbody>
</table>

Technology depreciation settings apply when calculating levelized costs. The interest rate and life times must be set for each technology with investment costs.
List of abstract base technology groups

Technologies must always define a parent, and this can either be one of the pre-defined abstract base technology groups or a user-defined group (see *Using tech_groups to group configuration*). The pre-defined groups are:

- **supply**: Supplies energy to a carrier, has a positive resource.
- **supply_plus**: Supplies energy to a carrier, has a positive resource. Additional possible constraints, including efficiencies and storage, distinguish this from *supply*.
- **demand**: Demands energy from a carrier, has a negative resource.
- **storage**: Stores energy.
- **transmission**: Transmits energy from one location to another.
- **conversion**: Converts energy from one carrier to another.
- **conversion_plus**: Converts energy from one or more carrier(s) to one or more different carrier(s).

A technology inherits the configuration that its parent group specifies (which, in turn, may inherit from its own parent).

**Note**: The identifiers of the abstract base tech groups are reserved and cannot be used for a user-defined technology or tech group.

The following lists the pre-defined base tech groups and the defaults they provide.

**supply**

Default constraints provided by the parent tech group:

```plaintext
esentials:
  parent:
  costs: {}
constraints:
  energy_prod: true
  resource: inf
  resource_unit: energy
```

Required constraints, allowed constraints, and allowed costs:

```plaintext
required_constraints:
  - [energy_cap_equals, energy_cap_max, energy_cap_per_unit]
allowed_costs:
  - depreciation_rate
  - energy_cap
  - export
  - interest_rate
  - om_annual
  - om_annual_investment_fraction
  - om_con
  - om_prod
  - purchase
  - resource_area
allowed_constraints:
  - energy_cap_equals
```

(continues on next page)
- energy_cap_equals_systemwide
- energy_cap_max
- energy_cap_max_systemwide
- energy_cap_min
- energy_cap_min_use
- energy_cap_per_unit
- energy_cap_scale
- energy_eff
- energy_prod
- energy_ramping
- export_cap
- export_carrier
- force_resource
- lifetime
- resource
- resource_area_equals
- resource_area_max
- resource_area_min
- resource_area_per_energy_cap
- resource_min_use
- resource_scale
- resource_unit
- units_equals
- units_equals_systemwide
- units_max
- units_max_systemwide
- units_min

supply_plus

Default constraints provided by the parent tech group:

essentials:
  parent:
  costs: {}
  constraints:
    energy_prod: true
    resource: inf
    resource_eff: 1.0
    resource_unit: energy

Required constraints, allowed constraints, and allowed costs:

required_constraints:
- [energy_cap_equals, energy_cap_max, energy_cap_per_unit]
allowed_costs:
- depreciation_rate
- energy_cap
- export
- interest_rate
- om_annual
- om_annual_investment_fraction
- om_con

(continues on next page)
om_prod
- purchase
- resource_area
- resource_cap
- storage_cap

allowed_constraints:
- charge_rate
- energy_cap_equals
- energy_cap_equals_systemwide
- energy_cap_max
- energy_cap_max_systemwide
- energy_cap_min
- energy_cap_min_use
- energy_cap_per_unit
- energy_cap_scale
- energy_eff
- energy_prod
- energy_ramping
- export_cap
- export_carrier
- force_resource
- lifetime
- parasitic_eff
- resource
- resource_area_equals
- resource_area_max
- resource_area_min
- resource_area_per_energy_cap
- resource_cap_equals
- resource_cap_equals_energy_cap
- resource_cap_max
- resource_cap_min
- resource_eff
- resource_min_use
- resource_scale
- resource_unit
- storage_cap_equals
- storage_cap_max
- storage_cap_min
- storage_cap_per_unit
- storage_initial
- storage_loss
- units_equals
- units_equals_systemwide
- units_max
- units_max_systemwide
- units_min

demand

Default constraints provided by the parent tech group:
essentials:
  parent:
  costs: {}
constraints:
  energy_con: true
  force_resource: true
  resource_unit: energy

Required constraints, allowed constraints, and allowed costs:

required_constraints:
  - resource
allowed_costs: []
allowed_constraints:
  - energy_con
  - force_resource
  - resource
  - resource_area_equals
  - resource_scale
  - resource_unit

storage

Default constraints provided by the parent tech group:

essentials:
  parent:
  costs: {}
constraints:
  energy_con: true
  energy_prod: true
  storage_cap_max: inf

Required constraints, allowed constraints, and allowed costs:

required_constraints:
  - [energy_cap_equals, energy_cap_max, energy_cap_per_unit]
  - [storage_cap_equals, storage_cap_max]
allowed_costs:
  - depreciation_rate
  - energy_cap
  - export
  - interest_rate
  - om_annual
  - om_annual_investment_fraction
  - om_prod
  - purchase
  - storage_cap
allowed_constraints:
  - charge_rate
  - energy_cap_equals
  - energy_cap_equals_systemwide
  - energy_cap_max
  - energy_cap_max_systemwide

(continues on next page)
transmission

Default constraints provided by the parent tech group:

```
essentials:
  parent:
 costs: {}
 constraints:
  energy_con: true
  energy_prod: true
```

Required constraints, allowed constraints, and allowed costs:

```
required_constraints:
  - [energy_cap_equals, energy_cap_max, energy_cap_per_unit]
allowed_costs:
  - depreciation_rate
  - energy_cap
  - energy_cap_per_distance
  - interest_rate
  - om_annual
  - om_annual_investment_fraction
  - om_prod
  - purchase
  - purchase_per_distance
allowed_constraints:
  - energy_cap_equals
  - energy_cap_max
  - energy_cap_per_unit
```

(continues on next page)
- energy_cap_scale
- energy_con
- energy_eff
- energy_eff_per_distance
- energy_prod
- lifetime
- one_way

conversion

Default constraints provided by the parent tech group:

```python
essentials:
    parent:
    costs: {}
constraints:
    energy_con: true
    energy_prod: true
```

Required constraints, allowed constraints, and allowed costs:

```python
required_constraints:
- [energy_cap_equals, energy_cap_max, energy_cap_per_unit]
allowed_costs:
- depreciation_rate
- energy_cap
- export
- interest_rate
- om_annual
- om_annual_investment_fraction
- om_con
- om_prod
- purchase
allowed_constraints:
- energy_cap_equals
- energy_cap_equals_systemwide
- energy_cap_max
- energy_cap_max_systemwide
- energy_cap_min
- energy_cap_min_use
- energy_cap_per_unit
- energy_cap_scale
- energy_con
- energy_eff
- energy_prod
- energy_ramping
- export_cap
- export_carrier
- lifetime
- units_equals
- units_equals_systemwide
- units_max
- units_max_systemwide
- units_min
```

1.7. More info
conversion_plus

Default constraints provided by the parent tech group:

```yaml
essentials:
  parent:
  costs: {}
constraints:
  energy_con: true
  energy_prod: true
```

Required constraints, allowed constraints, and allowed costs:

```yaml
required_constraints:
  - [energy_cap_equals, energy_cap_max, energy_cap_per_unit]
allowed_costs:
  - depreciation_rate
  - energy_cap
  - export
  - interest_rate
  - om_annual
  - om_annual_investment_fraction
  - om_con
  - om_prod
  - purchase
allowed_constraints:
  - carrier_ratios
  - energy_cap_equals
  - energy_cap_equals_systemwide
  - energy_cap_max
  - energy_cap_max_systemwide
  - energy_cap_min
  - energy_cap_min_use
  - energy_cap_per_unit
  - energy_cap_scale
  - energy_con
  - energy_eff
  - energy_prod
  - energy_ramping
  - export_cap
  - export_carrier
  - lifetime
  - units_equals
  - units_equals_systemwide
  - units_max
  - units_max_systemwide
  - units_min
```

**YAML configuration file format**

All configuration files (with the exception of time series data files) are in the YAML format, “a human friendly data serialisation standard for all programming languages”.

Configuration for Calliope is usually specified as `option: value` entries, where `value` might be a number, a text string, or a list (e.g. a list of further settings).
Calliope allows an abbreviated form for long, nested settings:

```yaml
one:
  two:
    three: x
```

can be written as:

```yaml
one.two.three: x
```

Calliope also allows a special `import:` directive in any YAML file. This can specify one or several YAML files to import. If both the imported file and the current file define the same option, the definition in the current file takes precedence.

Using quotation marks (" or ") to enclose strings is optional, but can help with readability. The three ways of setting `option` to `text` below are equivalent:

```yaml
option: "text"
option: 'text'
option: text
```

Sometimes, a setting can be either enabled or disabled, in this case, the boolean values `true` or `false` are used.

Comments can be inserted anywhere in YAML files with the `#` symbol. The remainder of a line after `#` is interpreted as a comment.

See the YAML website for more general information about YAML.

Calliope internally represents the configuration as `AttrDicts`, which are a subclass of the built-in Python dictionary data type (`dict`) with added functionality such as YAML reading/writing and attribute access to keys.

### 1.7.4 Mathematical formulation

This section details the mathematical formulation of the different components. For each component, a link to the actual implementing function in the Calliope code is given.

**Decision variables**

```python
calliope.backend.pyomo.variables.initialize_decision_variables(backend_model)
```

Defines decision variables.
### Variable Dimensions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy_cap</td>
<td>loc_techs</td>
</tr>
<tr>
<td>carrier_prod</td>
<td>loc_tech_carriers_prod, timesteps</td>
</tr>
<tr>
<td>carrier_con</td>
<td>loc_tech_carriers_con, timesteps</td>
</tr>
<tr>
<td>cost</td>
<td>costs, loc_techs_cost</td>
</tr>
<tr>
<td>resource_area</td>
<td>loc_techs_area</td>
</tr>
<tr>
<td>storage_cap</td>
<td>loc_techs_store</td>
</tr>
<tr>
<td>storage</td>
<td>loc_techs_store, timesteps</td>
</tr>
<tr>
<td>resource_con</td>
<td>loc_techs_supply_plus, timesteps</td>
</tr>
<tr>
<td>resource_cap</td>
<td>loc_techs_supply_plus</td>
</tr>
<tr>
<td>carrier_export</td>
<td>loc_tech_carriers_export, timesteps</td>
</tr>
<tr>
<td>cost_var</td>
<td>costs, loc_techs_om_cost, timesteps</td>
</tr>
<tr>
<td>cost_investment</td>
<td>costs, loc_techs_investment_cost</td>
</tr>
<tr>
<td>purchased</td>
<td>loc_techs_purchase</td>
</tr>
<tr>
<td>units</td>
<td>loc_techs_milp</td>
</tr>
<tr>
<td>operating_units</td>
<td>loc_techs_milp, timesteps</td>
</tr>
<tr>
<td>unused_supply</td>
<td>loc_carriers, timesteps</td>
</tr>
</tbody>
</table>

### Objective functions

**calliope.backend.pyomo.objective.minmax_cost_optimization** *(backend_model, cost_class, sense)*

Minimize or maximise total system cost for specified cost class.

If unmet_demand is in use, then the calculated cost of unmet_demand is added or subtracted from the total cost in the opposite sense to the objective.

\[
\min z = \sum_{loc::tech::cost} \text{cost} (loc::tech, \text{cost} = \text{cost}_k) + \sum_{loc::carrier,\text{ timestep}} \text{unmet_demand} (loc::carrier, \text{ timestep}) \times \text{bigM}
\]

**calliope.backend.pyomo.objective.check_feasibility** *(backend_model, **kwargs)*

Dummy objective, to check that there are no conflicting constraints.

\[
\min z = 1
\]

### Constraints

### Energy Balance

**calliope.backend.pyomo.constraints.energy_balance.system_balance_constraint_rule** *(backend_model, loc_carrier, timestep)*

System balance ensures that, within each location, the production and consumption of each carrier is balanced.

\[
\sum_{loc::tech::carrier::prod \in loc::carrier} \text{carrier::prod} (loc::tech::carrier, \text{ timestep}) + \sum_{loc::tech::carrier::con \in loc::carrier} \text{carrier::con} (loc::tech::carrier, \text{ timestep})
\]
calliope.backend.pyomo.constraints.energy_balance.balance_supply_constraint_rule(backend_model, loc_tech, timestep)

Limit production from supply techs to their available resource

\[
\text{min}_\text{use}(\text{loc} :: \text{tech}) \times \text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) \leq \frac{\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})}{\eta_{\text{energy}}(\text{loc} :: \text{tech}, \text{timestep})} \geq \text{available_resource}(\text{loc} :: \text{tech}, \text{timestep})
\]

If \(\text{force_resource}(\text{loc} :: \text{tech})\) is set:

\[
\frac{\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})}{\eta_{\text{energy}}(\text{loc} :: \text{tech}, \text{timestep})} = \text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) \quad \forall \text{loc} :: \text{tech} \in \text{techs}_{\text{supply}}
\]

Where:

\[
\text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) = \text{resource}(\text{loc} :: \text{tech}, \text{timestep}) \times \text{resource_scale}(\text{loc} :: \text{tech})
\]

if \(\text{loc} :: \text{tech}\) is in \(\text{loc} :: \text{techs}_{\text{area}}\):

\[
\text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) = \text{resource}(\text{loc} :: \text{tech}, \text{timestep}) \times \text{resource_scale}(\text{loc} :: \text{tech}) \times \text{resource}_{\text{area}}
\]

calliope.backend.pyomo.constraints.energy_balance.balance_demand_constraint_rule(backend_model, loc_tech, timestep)

Limit consumption from demand techs to their required resource.

\[
\text{carrier}_{\text{con}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \times \eta_{\text{energy}}(\text{loc} :: \text{tech}, \text{timestep}) \geq \text{required_resource}(\text{loc} :: \text{tech}, \text{timestep})
\]

If \(\text{force_resource}(\text{loc} :: \text{tech})\) is set:

\[
\text{carrier}_{\text{con}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \times \eta_{\text{energy}}(\text{loc} :: \text{tech}, \text{timestep}) = \text{required_resource}(\text{loc} :: \text{tech}, \text{timestep})
\]

Where:

\[
\text{required_resource}(\text{loc} :: \text{tech}, \text{timestep}) = \text{resource}(\text{loc} :: \text{tech}, \text{timestep}) \times \text{resource_scale}(\text{loc} :: \text{tech})
\]

if \(\text{loc} :: \text{tech}\) is in \(\text{loc} :: \text{techs}_{\text{area}}\):

\[
\text{required_resource}(\text{loc} :: \text{tech}, \text{timestep}) = \text{resource}(\text{loc} :: \text{tech}, \text{timestep}) \times \text{resource_scale}(\text{loc} :: \text{tech}) \times \text{resource}_{\text{area}}
\]

calliope.backend.pyomo.constraints.energy_balance.resource_availability_supply_plus_constraint_rule(backend_model, loc_tech, timestep)

Limit production from supply_plus techs to their available resource.

\[
\text{resource}_{\text{con}}(\text{loc} :: \text{tech}, \text{timestep}) \leq \text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) \quad \forall \text{loc} :: \text{tech} \in \text{techs}_{\text{supply}}, \forall \text{timestep}
\]

If \(\text{force_resource}(\text{loc} :: \text{tech})\) is set:

\[
\text{resource}_{\text{con}}(\text{loc} :: \text{tech}, \text{timestep}) = \text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) \quad \forall \text{loc} :: \text{tech} \in \text{techs}_{\text{supply}}, \forall \text{timestep}
\]

Where:

\[
\text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) = \text{resource}(\text{loc} :: \text{tech}, \text{timestep}) \times \text{resource_scale}(\text{loc} :: \text{tech})
\]

if \(\text{loc} :: \text{tech}\) is in \(\text{loc} :: \text{techs}_{\text{area}}\):

\[
\text{available_resource}(\text{loc} :: \text{tech}, \text{timestep}) = \text{resource}(\text{loc} :: \text{tech}, \text{timestep}) \times \text{resource_scale}(\text{loc} :: \text{tech}) \times \text{resource}_{\text{area}}(\text{loc})
\]
calliope.backend.pyomo.constraints.energy_balance.balance_transmission_constraint_rule(backend_model, loc_tech, timestep)

Balance carrier production and consumption of transmission technologies

\[-1 \times \text{carrier}_{\text{con}}(loc_{\text{from}} :: \text{tech} : loc_{\text{to}} :: \text{carrier}, \text{timestep}) \times \eta_{\text{energy}}(loc :: \text{tech}, \text{timestep}) = \text{carrier}_{\text{prod}}(loc_{\text{to}} :: \text{tech}) \]

Where a link is the connection between loc_{\text{from}} :: \text{tech} : loc_{\text{to}} and loc_{\text{to}} :: \text{tech} : loc_{\text{from}} for locations to and from.

calliope.backend.pyomo.constraints.energy_balance.balance_supply_plus_constraint_rule(backend_model, loc_tech, timestep)

Balance carrier production and resource consumption of supply_plus technologies alongside any use of resource storage.

\[
\text{storage}(loc :: \text{tech}, \text{timestep}) = \text{storage}(loc :: \text{tech}, \text{timestep}_{\text{previous}}) \times (1 - \text{storage-loss}(loc :: \text{tech}, \text{timestep}))^{\text{resolution}}
\]

If no storage is defined for the technology, this reduces to:

\[
\text{resource}_{\text{con}}(loc :: \text{tech}, \text{timestep}) \times \eta_{\text{resource}}(loc :: \text{tech}, \text{timestep}) = \frac{\text{carrier}_{\text{prod}}(loc :: \text{tech} :: \text{carrier}, \text{timestep})}{\eta_{\text{energy}}(loc :: \text{tech}, \text{timestep}) \times \eta_{\text{parasitic}}(loc :: \text{tech})}
\]

calliope.backend.pyomo.constraints.energy_balance.balance_storage_constraint_rule(backend_model, loc_tech, timestep)

Balance carrier production and consumption of storage technologies, alongside any use of the stored volume.

\[
\text{storage}(loc :: \text{tech}, \text{timestep}) = \text{storage}(loc :: \text{tech}, \text{timestep}_{\text{previous}}) \times (1 - \text{storage-loss}(loc :: \text{tech}, \text{timestep}))^{\text{resolution}}
\]

calliope.backend.pyomo.constraints.energy_balance.balance_storage_inter_cluster_rule(backend_model, loc_tech, datstep)

When clustering days, to reduce the timeseries length, balance the daily stored energy across all days of the original timeseries.

Ref: DOI 10.1016/j.apenergy.2018.01.023

\[
\text{storage}_{\text{inter-cluster}}(loc :: \text{tech}, \text{datestep}) = \text{storage}_{\text{inter-cluster}}(loc :: \text{tech}, \text{datestep}_{\text{previous}}) \times (1 - \text{storage-loss}(loc :: \text{tech}, \text{datestep}))^{\text{resolution}}
\]

Where timestep_{\text{final-cluster}}(datestep_{\text{previous}}) is the final timestep of the cluster in the clustered timeseries corresponding to the previous day.

calliope.backend.pyomo.constraints.energy_balance.storage_initial_rule(backend_model, loc_tech)

If storage is cyclic, allow an initial storage to still be set. This is applied to the storage of the final timestep/datestep of the series as that, in cyclic storage, is the ‘storage_previous_step’ for the first timestep/datestep.

If clustering and storage_inter_cluster exists:

\[
\text{storage}_{\text{inter-cluster}}(loc :: \text{tech}, \text{datestep}_{\text{final}}) \times ((1 - \text{storage-loss}) \times 24) = \text{storage}_{\text{initial}}(loc :: \text{tech}) \quad \forall loc :: \text{tech} \in \text{loc}_\text{tech}
\]

Where datestep_{\text{final}} is the last datestep of the timeseries.

Else: .. container:: scrolling-wrapper
storage(loc :: tech, timestepfinal) × ((1 − storage_toss) * 24) = storage_initial(loc :: tech) ∀loc :: tech ∈ loc :: techs

Where timestepfinal is the last timestep of the timeseries

**Capacity**

calliope.backend.pyomo.constraints.capacity.storage_capacity_constraint_rule(backend_model, loc_tech)

Set maximum storage capacity. Supply_plus & storage techs only

The first valid case is applied:

\[
\begin{align*}
storage_{cap}(loc :: tech) & = storage_{cap,equals}(loc :: tech), & \text{if} & & storage_{cap,equals}(loc :: tech) \\
& \leq storage_{cap,max}(loc :: tech), & \text{if} & & storage_{cap,max}(loc :: tech) \forall loc :: tech \in loc :: techsstore \\
& \text{unconstrained}, & \text{otherwise}
\end{align*}
\]

and (if equals not enforced):

\[
storage_{cap}(loc :: tech) \geq storage_{cap,min}(loc :: tech) \forall loc :: tech \in loc :: techsstore
\]

calliope.backend.pyomo.constraints.capacity.energy_capacity_storage_constraint_rule(backend_model, loc_tech)

Set an additional energy capacity constraint on storage technologies, based on their use of charge_rate.

\[
energy_{cap}(loc :: tech) \leq storage_{cap}(loc :: tech) \times charge_rate(loc :: tech) \forall loc :: tech \in loc :: techsstore
\]

calliope.backend.pyomo.constraints.capacity.resource_capacity_constraint_rule(backend_model, loc_tech)

Add upper and lower bounds for resource_cap.

The first valid case is applied:

\[
\begin{align*}
resource_{cap}(loc :: tech) & = resource_{cap,equals}(loc :: tech), & \text{if} & & resource_{cap,equals}(loc :: tech) \\
& \leq resource_{cap,max}(loc :: tech), & \text{if} & & resource_{cap,max}(loc :: tech) \forall loc :: tech \in loc :: techsstore \\
& \text{unconstrained}, & \text{otherwise}
\end{align*}
\]

and (if equals not enforced):

\[
resource_{cap}(loc :: tech) \geq resource_{cap,min}(loc :: tech) \forall loc :: tech \in loc :: techs_{finite_resource_supply_plus}
\]

calliope.backend.pyomo.constraints.capacity.resource_capacity_equals_energy_capacity_constraint_rule(backend_model, loc_tech)

Add equality constraint for resource_cap to equal energy_cap, for any technologies which have defined resource_cap_equals_energy_cap.

\[
resource_{cap}(loc :: tech) = energy_{cap}(loc :: tech) \forall loc :: tech \in loc :: techs_{finite_resource_supply_plus} \text{ if } resource_{cap}_equals_energy_cap
\]

calliope.backend.pyomo.constraints.capacity.resource_area_constraint_rule(backend_model, loc_tech)

Set upper and lower bounds for resource_area.
The first valid case is applied:

\[
\text{resource}_\text{area}(\text{loc} :: \text{tech}) = \begin{cases} 
\text{resource}_\text{area,equals}(\text{loc} :: \text{tech}), & \text{if resource}_\text{area,equals}(\text{loc} :: \text{tech}) \\
\leq \text{resource}_\text{area,max}(\text{loc} :: \text{tech}), & \text{if resource}_\text{area,max}(\text{loc} :: \text{tech}) \\
\end{cases} \quad \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}
\]

and (if equals not enforced):

\[
\text{resource}_\text{area}(\text{loc} :: \text{tech}) \geq \text{resource}_\text{area,min}(\text{loc} :: \text{tech}) \quad \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}
\]

calliope.backend.pyomo.constraints.capacity.\text{resource}_\text{area}_\text{per}_\text{energy}_\text{capacity}_\text{constraint}_\text{rule}

Add equality constraint for resource_area to equal a percentage of energy_cap, for any technologies which have defined resource_area_per_energy_cap

\[
\text{resource}_\text{area}(\text{loc} :: \text{tech}) = \text{energy}_\text{cap}(\text{loc} :: \text{tech}) \times \text{area}_\text{per}_\text{energy}_\text{cap}(\text{loc} :: \text{tech}) \quad \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}
\]

calliope.backend.pyomo.constraints.capacity.\text{resource}_\text{area}_\text{capacity}_\text{per}_\text{loc}_\text{constraint}_\text{rule}

Set upper bound on use of area for all locations which have available_area constraint set. Does not consider resource_area applied to demand technologies

\[
\sum_{\text{tech}} \text{resource}_\text{area}(\text{loc} :: \text{tech}) \leq \text{available}_\text{area} \quad \forall \text{loc} \in \text{loc} \text{ if available}_\text{area}(\text{loc})
\]

calliope.backend.pyomo.constraints.capacity.\text{energy}_\text{capacity}_\text{constraint}_\text{rule}(\text{backend}_\text{model}, \text{loc}_\text{tech})

Set upper and lower bounds for energy_cap.

The first valid case is applied:

\[
\text{energy}_\text{cap}(\text{loc} :: \text{tech}) = \begin{cases} 
\text{energy}_\text{cap,equals}(\text{loc} :: \text{tech}), & \text{if energy}_\text{cap,equals}(\text{loc} :: \text{tech}) \\
\leq \text{energy}_\text{cap,max}(\text{loc} :: \text{tech}), & \text{if energy}_\text{cap,max}(\text{loc} :: \text{tech}) \\
\end{cases} \quad \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}
\]

and (if equals not enforced):

\[
\frac{\text{energy}_\text{cap}(\text{loc} :: \text{tech})}{\text{energy}_\text{cap, scale}(\text{loc} :: \text{tech})} \geq \text{energy}_\text{cap,min}(\text{loc} :: \text{tech}) \quad \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}
\]

calliope.backend.pyomo.constraints.capacity.\text{energy}_\text{capacity}_\text{systemwide}_\text{constraint}_\text{rule}(\text{backend}_\text{model}, \text{loc}_\text{tech})

Set constraints to limit the capacity of a single technology type across all locations in the model.

The first valid case is applied:

\[
\sum_{\text{loc}} \text{energy}_\text{cap}(\text{loc} :: \text{tech}) = \begin{cases} 
\text{energy}_\text{cap,equals,systemwide}(\text{loc} :: \text{tech}), & \text{if energy}_\text{cap,equals,systemwide}(\text{loc} :: \text{tech}) \\
\leq \text{energy}_\text{cap,max,systemwide}(\text{loc} :: \text{tech}), & \text{if energy}_\text{cap,max,systemwide}(\text{loc} :: \text{tech}) \\
\end{cases} \quad \forall \text{tech} \in \text{techs}
\]

\textit{Dispatch}

calliope.backend.pyomo.constraints.dispatch.\text{carrier}_\text{production}_\text{max}_\text{constraint}_\text{rule}(\text{backend}_\text{model}, \text{loc}_\text{tech}_\text{carrier}, \text{timestep})

Set maximum carrier production. All technologies.

\[
\text{carrier}_\text{prod}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \leq \text{energy}_\text{cap}(\text{loc} :: \text{tech}) \times \text{timestep}_\text{resolution}(\text{timestep}) \times \text{parasitic}_\text{eff}
\]
Set minimum carrier production. All technologies except conversion_plus.

\[
\text{carrier}_{\text{prod}}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}) \geq \text{energy}_{\text{cap}}(\text{loc}::\text{tech}) \times \text{timestep}_{\text{resolution}}(\text{timestep}) \times \text{energy}_{\text{cap}, \text{min}}
\]

Set maximum carrier consumption for demand, storage, and transmission techs.

\[
\text{carrier}_{\text{con}}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}) \geq -1 \times \text{energy}_{\text{cap}}(\text{loc}::\text{tech}) \times \text{timestep}_{\text{resolution}}(\text{timestep})
\]

Set maximum resource consumed by supply_plus techs.

\[
\text{resource}_{\text{con}}(\text{loc}::\text{tech}, \text{timestep}) \leq \text{timestep}_{\text{resolution}}(\text{timestep}) \times \text{resource}_{\text{cap}}(\text{loc}::\text{tech})
\]

Set maximum stored energy. Supply_plus & storage techs only.

\[
\text{storage}(\text{loc}::\text{tech}, \text{timestep}) \leq \text{storage}_{\text{cap}}(\text{loc}::\text{tech})
\]

Ramping up constraint.

\[
\text{diff}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}) \leq \text{max}_{\text{ramping rate}}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep})
\]

Ramping down constraint.

\[
-1 \times \text{max}_{\text{ramping rate}}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}) \leq \text{diff}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep})
\]

Ramping rate constraints.

Direction: 0 is up, 1 is down.

\[
\text{diff}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}) = (\text{carrier}_{\text{prod}}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}) + \text{carrier}_{\text{con}}(\text{loc}::\text{tech}::\text{carrier}, \text{timestep}))
\]
When clustering days, to reduce the timeseries length, set limits on intra-cluster auxiliary maximum storage decision variable. Ref: DOI 10.1016/j.apenergy.2018.01.023

\[
\text{storage}(\text{loc} :: \text{tech}, \text{timestep}) \leq \text{storage}_{\text{intra-cluster, max}}(\text{loc} :: \text{tech}, \text{cluster}(\text{timestep})) \quad \forall \text{loc} :: \text{tech} \in \text{techs}_{\text{storage}}
\]

Where \(\text{cluster}(\text{timestep})\) is the cluster number in which the timestep is located.

When clustering days, to reduce the timeseries length, set limits on intra-cluster auxiliary minimum storage decision variable. Ref: DOI 10.1016/j.apenergy.2018.01.023

\[
\text{storage}(\text{loc} :: \text{tech}, \text{timestep}) \geq \text{storage}_{\text{intra-cluster, min}}(\text{loc} :: \text{tech}, \text{cluster}(\text{timestep})) \quad \forall \text{loc} :: \text{tech} \in \text{techs}_{\text{storage}}
\]

Where \(\text{cluster}(\text{timestep})\) is the cluster number in which the timestep is located.

When clustering days, to reduce the timeseries length, set maximum limit on the intra-cluster and inter-date stored energy. intra-cluster = all timesteps in a single cluster datesteps = all dates in the unclustered timeseries (each has a corresponding cluster) Ref: DOI 10.1016/j.apenergy.2018.01.023

\[
\text{storage}_{\text{inter-cluster}}(\text{loc} :: \text{tech}, \text{datestep}) + \text{storage}_{\text{intra-cluster, max}}(\text{loc} :: \text{tech}, \text{cluster}(\text{datestep})) \leq \text{storage}_{\text{cap}}(\text{loc} :: \text{tech})
\]

Where \(\text{cluster}(\text{datestep})\) is the cluster number in which the datestep is located.

When clustering days, to reduce the timeseries length, set minimum limit on the intra-cluster and inter-date stored energy. intra-cluster = all timesteps in a single cluster datesteps = all dates in the unclustered timeseries (each has a corresponding cluster) Ref: DOI 10.1016/j.apenergy.2018.01.023

\[
\text{storage}_{\text{inter-cluster}}(\text{loc} :: \text{tech}, \text{datestep}) \times (1 - \text{storage}_{\text{loss}}(\text{loc} :: \text{tech}, \text{timestep}))^{24} + \text{storage}_{\text{intra-cluster, min}}(\text{loc} :: \text{tech}, \text{cluster}(\text{datestep})) \geq 0
\]

Where \(\text{cluster}(\text{datestep})\) is the cluster number in which the datestep is located.

Costs

Combine investment and time varying costs into one cost per technology

\[
\text{cost}(\text{cost}, \text{loc} :: \text{tech}) = \text{cost}_{\text{investment}}(\text{cost}, \text{loc} :: \text{tech}) + \sum_{\text{timestep} \in \text{timesteps}} \text{cost}_{\text{var}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep})
\]
Transmission technologies “exist” at two locations, so their cost is divided by 2.

\[
cost_{\text{con}}(\text{cost}, \text{loc} :: \text{tech}) = \text{depreciation\_rate} \times \text{ts\_weight} \times (\text{cost}_{\text{energy\_cap}}(\text{cost}, \text{loc} :: \text{tech}) \times \text{energy\_cap}(\text{loc} :: \text{tech}) + \text{cost})
\]

\[
\text{cost}_{\text{var}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) + \text{cost}_{\text{con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep})
\]

\[
\text{cost}_{\text{prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{om\_prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) \times \text{weight}(\text{timestep}) \times \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech}, \text{timestep})
\]

\[
\text{prod\_con\_eff} = \begin{cases} 
\text{resource}_{\text{con}}(\text{loc} :: \text{tech}, \text{timestep}), & \text{if \ loc :: \ tech} \in \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech}, \text{timestep}) \\ 
\frac{\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})}{\text{energy\_ff}(\text{loc} :: \text{tech}, \text{timestep})}, & \text{if \ loc :: \ tech} \in \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech}, \text{timestep})
\end{cases}
\]

\[
\text{cost}_{\text{con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{om\_con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) \times \text{weight}(\text{timestep})
\]

Export

\[
\text{calliope.backend.pyomo.constraints.costs.cost\_var\_constraint\_rule}(\text{backend\_model}, \text{cost}, \text{loc\_tech}, \text{timestep})
\]

Calculate costs from time-varying decision variables

\[
\text{cost}_{\text{var}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) + \text{cost}_{\text{con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep})
\]

\[
\text{cost}_{\text{prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{om\_prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) \times \text{weight}(\text{timestep}) \times \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech}, \text{timestep})
\]

\[
\text{prod\_con\_eff} = \begin{cases} 
\text{resource}_{\text{con}}(\text{loc} :: \text{tech}, \text{timestep}), & \text{if \ loc :: \ tech} \in \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech}, \text{timestep}) \\ 
\frac{\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})}{\text{energy\_ff}(\text{loc} :: \text{tech}, \text{timestep})}, & \text{if \ loc :: \ tech} \in \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech}, \text{timestep})
\end{cases}
\]

\[
\text{cost}_{\text{con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{om\_con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) \times \text{weight}(\text{timestep})
\]

Update system balance constraint (from energy\_balance.py) to include export

Math given in system\_balance\_constraint\_rule()

\[
\text{calliope.backend.pyomo.constraints.export.update\_system\_balance\_constraint}(\text{backend\_model}, \text{loc\_carrier}, \text{timestep})
\]

Ensure no technology can ‘pass’ its export capability to another technology with the same carrier\_out, by limiting its export to the capacity of its production

\[
\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \geq \text{carrier}_{\text{export}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \forall \text{loc} :: \text{tech} :: \text{carrier} \in \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})
\]

\[
\text{calliope.backend.pyomo.constraints.export.update\_costs\_var\_constraint}(\text{backend\_model}, \text{cost}, \text{loc\_tech}, \text{timestep})
\]

Update time varying cost constraint (from costs.py) to include export

\[
\text{cost}_{\text{var}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) = \text{cost}_{\text{prod}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) + \text{cost}_{\text{con}}(\text{cost}, \text{loc} :: \text{tech}, \text{timestep}) + \text{carrier}_{\text{export}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})
\]

\[
\text{calliope.backend.pyomo.constraints.export.export\_max\_constraint\_rule}(\text{backend\_model}, \text{loc\_tech\_carrier}, \text{timestep})
\]

Set maximum export. All exporting technologies.

\[
\text{carrier}_{\text{export}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \leq \text{export}_{\text{cap}}(\text{loc} :: \text{tech}) \forall \text{loc} :: \text{tech} :: \text{carrier} \in \text{loc\_carrier} :: \text{tech}\_\text{carrier}
\]
If the technology is defined by integer units, not a continuous capacity, this constraint becomes:

\[
\text{carrier}_{\text{export}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{times} \text{tep}) \leq \text{export}_{\text{cap}}(\text{loc} :: \text{tech}) \times \text{operating}_{\text{units}}(\text{loc} :: \text{tech}, \text{times} \text{tep})
\]

**MILP**

\text{calliope.backend.pyomo.constraints.milp}\_\text{unit} \_\text{commitment} \_\text{constraint} \_\text{rule} (\text{backend} _\text{model}, \text{loc} _\text{tech}, \text{times} \text{tep})

Constraining the number of integer units \text{operating}_{\text{units}}(\text{loc} :: \text{tech}, \text{times} \text{tep}) of a technology which can operate in a given timestep, based on maximum purchased units \text{units}(\text{loc} :: \text{tech})

\text{operating}_{\text{units}}(\text{loc} :: \text{tech}, \text{times} \text{tep}) \leq \text{units}(\text{loc} :: \text{tech}) \ \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}_{\text{milp}}, \forall \text{times} \text{tep} \in \text{times} \text{steps}

\text{calliope.backend.pyomo.constraints.milp}\_\text{unit} \_\text{capacity} \_\text{constraint} \_\text{rule} (\text{backend} _\text{model}, \text{loc} _\text{tech})

Add upper and lower bounds for purchased units of a technology

\[
\text{units}(\text{loc} :: \text{tech}) = \begin{cases} 
\text{units}_{\text{equals}}(\text{loc} :: \text{tech}), & \text{if } \text{units}_{\text{equals}}(\text{loc} :: \text{tech}) \\
\leq \text{units}_{\text{max}}(\text{loc} :: \text{tech}), & \text{if } \text{units}_{\text{max}}(\text{loc} :: \text{tech}) \\
\text{unconstrained}, & \text{otherwise}
\end{cases}
\]

and (if equals not enforced):

\[
\text{units}(\text{loc} :: \text{tech}) \geq \text{units}_{\text{min}}(\text{loc} :: \text{tech}) \ \forall \text{loc} :: \text{tech} \in \text{loc} :: \text{techs}_{\text{milp}}
\]

\text{calliope.backend.pyomo.constraints.milp}\_\text{carrier} \_\text{production} \_\text{max} \_\text{milp} \_\text{constraint} \_\text{rule} (\text{backend} _\text{model}, \text{loc} _\text{tech} _\text{times} \text{tep})

Set maximum carrier production of MILP techs that aren’t conversion plus

\[
\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{times} \text{tep}) \leq \text{energy}_{\text{cap}, \text{per} \text{unit}}(\text{loc} :: \text{tech}) \times \text{times} \text{ep} \text{resolution}(\text{times} \text{tep}) \times \text{operating}_{\text{units}}(\text{loc} :: \text{tech}) \times \eta_{\text{parasitic}}
\]

\eta_{\text{parasitic}} is only activated for \text{supply} \_\text{plus} technologies

\text{calliope.backend.pyomo.constraints.milp}\_\text{carrier} \_\text{production} \_\text{max} \_\text{conversion} \_\text{plus} \_\text{milp} \_\text{constraint} \_\text{rule}

Set maximum carrier production of conversion\_plus MILP techs

\[
\sum_{\text{loc} :: \text{tech} :: \text{carrier} \in \text{loc} :: \text{tech} :: \text{carriers}_{\text{out}}} \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{times} \text{tep}) \leq \text{energy}_{\text{cap}, \text{per} \text{unit}}(\text{loc} :: \text{tech}) \times \text{times} \text{ep} \text{resolution}(\text{times} \text{tep}) \times \text{operating}_{\text{units}}(\text{loc} :: \text{tech})
\]

\text{calliope.backend.pyomo.constraints.milp}\_\text{carrier} \_\text{production} \_\text{min} \_\text{milp} \_\text{constraint} \_\text{rule} (\text{backend} _\text{model}, \text{loc} _\text{tech} _\text{times} \text{tep})

Set minimum carrier production of MILP techs that aren’t conversion plus

\[
\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{times} \text{tep}) \geq \text{energy}_{\text{cap}, \text{per} \text{unit}}(\text{loc} :: \text{tech}) \times \text{times} \text{ep} \text{resolution}(\text{times} \text{tep}) \times \text{operating}_{\text{units}}(\text{loc} :: \text{tech})
\]

\text{calliope.backend.pyomo.constraints.milp}\_\text{carrier} \_\text{production} \_\text{min} \_\text{conversion} \_\text{plus} \_\text{milp} \_\text{constraint} \_\text{rule}

Set minimum carrier production of conversion\_plus MILP techs

\[
\sum_{\text{loc} :: \text{tech} :: \text{carrier} \in \text{loc} :: \text{tech} :: \text{carriers}_{\text{out}}} \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{times} \text{tep}) \geq \text{energy}_{\text{cap}, \text{per} \text{unit}}(\text{loc} :: \text{tech}) \times \text{times} \text{ep} \text{resolution}(\text{times} \text{tep}) \times \text{operating}_{\text{units}}(\text{loc} :: \text{tech})
\]
The first valid case is applied:

\[
\begin{align*}
\text{energy}_{\text{cap}}(loc :: tech) &= \text{energy}_{\text{cap.scale}}(loc :: tech) \\
&\begin{cases}
= \text{energy}_{\text{cap.equals}}(loc :: tech) \times \text{purchased}(loc :: tech), & \text{if } \text{energy}_{\text{cap.equals}}(loc :: tech) \\
\leq \text{energy}_{\text{cap.max}}(loc :: tech) \times \text{purchased}(loc :: tech), & \text{if } \text{energy}_{\text{cap.max}}(loc :: tech) \\
\text{unconstrained}, & \text{otherwise}
\end{cases}
\end{align*}
\]

Set minimum energy capacity decision variable upper bound as a function of binary purchase variable

\[
\frac{\text{energy}_{\text{cap}}(loc :: tech)}{\text{energy}_{\text{cap.scale}}(loc :: tech)} \geq \text{energy}_{\text{cap.min}}(loc :: tech) \times \text{purchased}(loc :: tech) \quad \forall loc :: tech \in \text{techs}
\]

Set maximum storage capacity.

The first valid case is applied:

\[
\begin{align*}
\text{storage}_{\text{cap}}(loc :: tech) &= \text{storage}_{\text{cap.equals}}(loc :: tech) \times \text{purchased}, & \text{if } \text{storage}_{\text{cap.equals}}(loc :: tech) \\
&\leq \text{storage}_{\text{cap.max}}(loc :: tech) \times \text{purchased}, & \text{if } \text{storage}_{\text{cap.max}}(loc :: tech) \\
\text{unconstrained}, & \text{otherwise}
\end{align*}
\]

Set minimum storage capacity decision variable as a function of binary purchase variable

if equals not enforced for storage_cap:

\[
\text{storage}_{\text{cap}}(loc :: tech) \geq \text{storage}_{\text{cap.min}}(loc :: tech) \times \text{purchased}(loc :: tech) \quad \forall loc :: tech \in \text{techs}_{\text{purchase,store}}
\]
calliope.backend.pyomo.constraints.milp.update_costs_investment_units_constraint (backend_model, cost, loc_tech)

Add MILP investment costs (cost * number of units purchased)

\[ \text{cost}_{\text{investment}}(\text{cost}, \text{loc} :: \text{tech}) + = \text{units}(
\text{loc} :: \text{tech}) \times \text{cost}_{\text{purchase}}(\text{cost}, \text{loc} :: \text{tech}) \times \text{timestep}_{\text{weight}} \times \text{depreciation} \]

calliope.backend.pyomo.constraints.milp.update_costs_investment_purchase_constraint (backend_model, cost, loc_tech)

Add binary investment costs (cost * binary_purchased_unit)

\[ \text{cost}_{\text{investment}}(\text{cost}, \text{loc} :: \text{tech}) + = \text{purchased}(
\text{loc} :: \text{tech}) \times \text{cost}_{\text{purchase}}(\text{cost}, \text{loc} :: \text{tech}) \times \text{timestep}_{\text{weight}} \times \text{depreciation} \]

calliope.backend.pyomo.constraints.milp.unit_capacity_systemwide_constraint_rule (backend_model, tech)

Set constraints to limit the number of purchased units of a single technology type across all locations in the model.

The first valid case is applied:

\[ \sum_{\text{loc}} \text{units}(
\text{loc} :: \text{tech}) + \text{purchased}(
\text{loc} :: \text{tech}) \begin{cases} = \text{units}_{\text{equals}, \text{systemwide}}(\text{tech}), & \text{if} \text{units}_{\text{equals}, \text{systemwide}}(\text{tech}) \\ \leq \text{units}_{\text{max}, \text{systemwide}}(\text{tech}), & \text{if} \text{units}_{\text{max}, \text{systemwide}}(\text{tech}) \quad \forall \text{tech} \\ \text{unconstrained}, & \text{otherwise} \end{cases} \]

Conversion

calliope.backend.pyomo.constraints.conversion.balance_conversion_constraint_rule (backend_model, loc_tech, timestep)

Balance energy carrier consumption and production

\[ -1 \times \text{carrier}_{\text{con}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \times \eta_{\text{energy}}(\text{loc} :: \text{tech}, \text{timestep}) = \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \]

calliope.backend.pyomo.constraints.conversion.cost_var_conversion_constraint_rule (backend_model, cost, loc_tech, timestep)

Add time-varying conversion technology costs

\[ \text{cost}_{\text{var}}(\text{loc} :: \text{tech}, \text{cost}, \text{timestep}) = \text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep}) \times \text{timestep}_{\text{weight}}(\text{timestep}) \times \text{cost}_{\text{om}} \]

Conversion_plus

calliope.backend.pyomo.constraints.conversion_plus.balance_conversion_plus_primary_constraint_rule (backend_model, cost, loc_tech, carriers_in, carriers_out, timestep)

Balance energy carrier consumption and production for carrier_in and carrier_out

\[ \sum_{\text{loc} :: \text{tech} :: \text{carrier} @ \text{loc} :: \text{tech} :: \text{carriers}_{\text{out}}} \frac{\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})}{\text{carrier}_{\text{ratio}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{'out'})} = -1 \sum_{\text{loc} :: \text{tech} :: \text{carrier} @ \text{loc} :: \text{tech} :: \text{carriers}_{\text{in}}} \frac{\text{carrier}_{\text{prod}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{timestep})}{\text{carrier}_{\text{ratio}}(\text{loc} :: \text{tech} :: \text{carrier}, \text{'in'})} \]

86 Chapter 1. User guide
Set maximum conversion_plus carrier production.

\[
\sum_{loc::tech::carrier\in loc::tech::carriers_{out}} \text{carrier}_{\text{prod}}(loc :: tech :: carrier, \text{timestep}) \leq \text{energy}_{\text{cap}}(loc :: tech) \times \text{timestep}_{\text{res}}
\]

Set minimum conversion_plus carrier production.

\[
\sum_{loc::tech::carrier\in loc::tech::carriers_{out}} \text{carrier}_{\text{prod}}(loc :: tech :: carrier, \text{timestep}) \leq \text{energy}_{\text{cap}}(loc :: tech) \times \text{timestep}_{\text{res}}
\]

Add time-varying conversion_plus technology costs

\[
\text{cost}_{\text{var}}(loc :: tech, \text{cost}, \text{timestep}) = \text{carrier}_{\text{prod}}(loc :: tech :: \text{carrier}_{\text{primary}}, \text{timestep}) \times \text{timestep}_{\text{weight}}(\text{timestep}) \times \text{cost}_{\text{om}, \text{con}}(loc :: tech, \text{carrier}, \text{timestep})
\]

Force all carrier_in_2/carrier_in_3 and carrier_out_2/carrier_out_3 to follow carrier_in and carrier_out (respectively).

If \(tier\) in \('[\text{out}_2', '\text{out}_3']\):

\[
\sum_{loc::tech::carrier\in loc::tech::carriers_{out}} \left( \frac{\text{carrier}_{\text{prod}}(loc :: tech :: carrier, \text{timestep})}{\text{carrier}_{\text{ratio}}(loc :: tech :: carrier, '\text{out}')} \right) = \sum_{loc::tech::carrier\in loc::tech::carriers_{\text{tier}}} \left( \frac{\text{carrier}_{\text{con}}(loc :: tech :: carrier, \text{timestep})}{\text{carrier}_{\text{ratio}}(loc :: tech :: carrier, '\text{out}')} \right)
\]

If \(tier\) in \('[\text{in}_2', '\text{in}_3']\):

\[
\sum_{loc::tech::carrier\in loc::tech::carriers_{in}} \left( \frac{\text{carrier}_{\text{con}}(loc :: tech :: carrier, \text{timestep})}{\text{carrier}_{\text{ratio}}(loc :: tech :: carrier, '\text{in}')} \right) = \sum_{loc::tech::carrier\in loc::tech::carriers_{\text{tier}}} \left( \frac{\text{carrier}_{\text{con}}(loc :: tech :: carrier, \text{timestep})}{\text{carrier}_{\text{ratio}}(loc :: tech :: carrier, '\text{in}')} \right)
\]

**Network**

\text{calliope.backend.pyomo.constraints.network.symmetric_transmission_constraint_rule}(\text{backend}\_\text{model}, \text{loc}\_\text{tech})

Constrain \(e\_\text{cap}\) symmetrically for transmission nodes. Transmission techs only.

\[
\text{energy}_{\text{cap}}(\text{loc1} :: \text{tech} : \text{loc2}) = \text{energy}_{\text{cap}}(\text{loc2} :: \text{tech} : \text{loc1})
\]

**Policy**

\text{calliope.backend.pyomo.constraints.policy.group_share_energy_cap_constraint_rule}(\text{backend}\_\text{model}, \text{tech}\_\text{list}, \text{what})

Enforce shares in \(e\_\text{cap}\) for groups of technologies. Applied to \(\text{supply}\) and \(\text{supply}\_\text{plus}\) technologies.
\[
\sum_{loc::tech\in\text{given\_group}} \text{energy}_{\text{cap}}(loc :: tech) = \text{fraction} \times \sum_{loc::tech\in\text{loc\_techs}\_\text{supply} \cup \text{loc\_techs}\_\text{supply\_plus}} \text{energy}_{\text{cap}}(loc :: tech)
\]

\text{calliope.backend.pyomo.constraints.policy.group\_share\_carrier\_prod\_constraint\_rule}(\text{backend\_model}, \text{tech\_list\_carrier}, \text{what})

Enforce shares in carrier\_prod for groups of technologies. Applied to \text{loc\_tech\_carriers\_supply\_all}, which includes supply, supply\_plus, conversion, and conversion\_plus.

\[
\sum_{loc::tech::carrier\in\text{given\_group}, timestep\in\text{timesteps}} \text{carrier}_{\text{prod}}(loc :: tech :: carrier, timestep) = \text{fraction} \times \sum_{loc::tech::carrier\in\text{loc\_tech\_carriers\_supply\_all}} \text{carrier}_{\text{prod}}(loc :: tech :: carrier, timestep)
\]

\text{calliope.backend.pyomo.constraints.policy.reserve\_margin\_constraint\_rule}(\text{backend\_model}, \text{carrier})

Enforces a system reserve margin per carrier.

\[
\sum_{loc::tech::carrier\in\text{loc\_tech\_carriers\_supply\_all}} \text{energy}_{\text{cap}}(loc :: tech :: carrier, timestep_{\text{max\_demand}}) \geq \sum_{loc::tech::carrier\in\text{loc\_tech\_carriers\_supply\_all}} \text{energy}_{\text{cap}}(loc :: tech :: carrier, timestep_{\text{max\_demand}})
\]

1.8 Development guide

Contributions are very welcome! See our contributors guide on GitHub for information on how to contribute.

The code lives on GitHub at calliope-project/calliope. Development takes place in the master branch. Stable versions are tagged off of master with semantic versioning.

Tests are included and can be run with \texttt{py.test} from the project’s root directory.

Also see the list of open issues, planned milestones and projects for an overview of where development is heading, and join us on Gitter to ask questions or discuss code.

1.8.1 Installing a development version

As when installing a stable version, using \texttt{conda} is recommended.

If you only want to track the latest commit, without having a local Calliope repository, then just download the base.yml and latest.yml requirements files and run (assuming both are saved into a directory called requirements):

\[
\texttt{\$ conda env create -n calliope_latest \_--file=requirements/base.yml \_--file=requirements/latest.yml}
\]

This will create a conda environment called calliope_latest.

To actively contribute to Calliope development, you’ll instead want to clone the repository, giving you an editable copy. This will provide you with the master branch in a known location on your local device.

First, clone the repository:

\[
\texttt{\$ git clone https://github.com/calliope-project/calliope}
\]

Using Anaconda/conda, install all requirements, including the free and open source GLPK solver, into a new environment, e.g. calliope_dev:
1.8.2 Creating modular extensions

As of version 0.6.0, dynamic loading of custom constraint generator extensions has been removed due it not not being used by users of Calliope. The ability to dynamically load custom functions to adjust time resolution remains (see below).

Time functions and masks

Custom functions that adjust time resolution can be loaded dynamically during model initialisation. By default, Calliope first checks whether the name of a function or time mask refers to a function from the `calliope.core.time.masks` or `calliope.core.time.funcs` module, and if not, attempts to load the function from an importable module:

```python
time:
masks:
    - {function: week, options: {day_func: 'extreme', tech: 'wind', how: 'min'}}
    - {function: my_custom_module.my_custom_mask, options: {...}}

function: my_custom_module.my_custom_function
function_options: {...}
```

1.8.3 Profiling

To profile a Calliope run with the built-in national-scale example model, then visualise the results with snakeviz:

```bash
make profile  # will dump profile output in the current directory
snakeviz calliope.profile  # launch snakeviz to visually examine profile
```

Use mprof plot to plot memory use.

Other options for visualising:

- Interactive visualisation with KCachegrind (on macOS, use QCachegrind, installed e.g. with `brew install qcachegrind`)

  ```bash
  pyprof2calltree -i calliope.profile -o calliope.calltree
  kcachegrind calliope.calltree
  ```

- Generate a call graph from the call tree via graphviz

  ```bash
  # brew install gprof2dot
  gprof2dot -f callgrind calliope.calltree | dot -Tsvg -o callgraph.svg
  ```
1.8.4 Checklist for new release

Pre-release

- Make sure all unit tests pass
- Build up-to-date Plotly plots for the documentation with (make doc-plots)
- Re-run tutorial Jupyter notebooks, found in doc/_static/notebooks
- Make sure documentation builds without errors
- Make sure the release notes are up-to-date, especially that new features and backward incompatible changes are clearly marked

Create release

- Change _version.py version number
- Update changelog with final version number and release date
- Commit with message “Release vXXXX”, then add a “vXXXX” tag, push both to GitHub
- Create a release through the GitHub web interface, using the same tag, titling it “Release vXXXX” (required for Zenodo to pull it in)
- Upload new release to PyPI: make all-dist
- Update the conda-forge package:
  - Fork conda-forge/calliope-feedstock, and update recipe/meta.yaml with:
    * Version number: {% set version = "XXXX" %}
    * SHA256 of latest version from PyPI: {% set sha256 = "XXXX" %}
    * Reset build: number: 0 if it is not already at zero
    * If necessary, carry over any changed requirements from setup.py or requirements/base.yml
  - Submit a pull request from an appropriately named branch in your fork (e.g. vXXXX) to the conda-forge/calliope-feedstock repository

Post-release

- Update changelog, adding a new vXXXX-dev heading, and update _version.py accordingly, in preparation for the next master commit
- Update the calliope_version setting in all example models to match the new version, but without the -dev string (so 0.6.0-dev is 0.6.0 for the example models)

Note: Adding ‘-dev’ to the version string, such as __version__ = '0.1.0-dev', is required for the custom code in doc/conf.py to work when building in-development versions of the documentation.
CHAPTER 2

API documentation

Documents functions, classes and methods:

2.1 API Documentation

2.1.1 Model class

class calliope.Model(config, model_data=None, *args, **kwargs)
A Calliope Model.

save_commented_model_yaml(path)
Save a fully built and commented version of the model to a YAML file at the given path. Comments in the file indicate where values were overridden. This is Calliope’s internal representation of a model directly before the model_data xarray.Dataset is built, and can be useful for debugging possible issues in the model formulation.

run(force_rerun=False, **kwargs)
Run the model. If force_rerun is True, any existing results will be overwritten.

Additional kwargs are passed to the backend.

get_formatted_array(var)
Return an xr.DataArray with locs, techs, and carriers as separate dimensions.

Parameters

var [str] Decision variable for which to return a DataArray.

to_netcdf(path)
Save complete model data (inputs and, if available, results) to a NetCDF file at the given path.

to_csv(path, dropna=True)
Save complete model data (inputs and, if available, results) as a set of CSV files to the given path.

Parameters
**2.1.2 Time series**

```
calliope.core.time.clustering.get_clusters(data, func, timesteps_per_day, tech=None,
    **kwargs)
```

Run a clustering algorithm on the timeseries data supplied. All timeseries data is reshaped into one row per day before clustering into similar days.

**Parameters**
- `data` [xarray.Dataset] Should be normalized
- `func` [str] ‘kmeans’ or ‘hierarchical’ for KMeans or Agglomerative clustering, respectively
- `timesteps_per_day` [int] Total number of timesteps in a day
- `tech` [list, optional] list of strings referring to technologies by which clustering is undertaken. If none (default), all technologies within timeseries variables will be used.
- `timesteps` [list or str, optional] Subset of the time domain within which to apply clustering.
- `k` [int, optional] Number of clusters to create. If none (default), will use Hartigan’s rule to infer a reasonable number of clusters.
- `variables` [list, optional] data variables (e.g. `resource`, `energy_eff`) by whose values the data will be clustered. If none (default), all timeseries variables will be used.

**Returns**
- `clusters` [dataframe] Indexed by timesteps and with locations as columns, giving cluster membership for first timestep of each day.
- `clustered_data` [sklearn.cluster object] Result of clustering using sklearn.KMeans(k).fit(X) or sklearn.KMeans(k).AgglomerativeClustering(X). Allows user to access specific attributes, for detailed statistical analysis.

```
calliope.core.time.masks.extreme(data, tech, var='resource', how='max', length='1D',
    groupby_length=None, padding=None, normalize=True, **kwargs)
```

Returns timesteps for period of length where var for the technology tech across the given list of locations is either minimal or maximal.

**Parameters**
- `data` [xarray.Dataset]
- `tech` [str] Technology whose `var` to find extreme for.
- `var` [str, optional] default ‘resource’
- `how` [str, optional] ‘max’ (default) or ‘min’.
- `length` [str, optional] Defaults to ‘1D’.
n [int, optional] Number of periods of length to look for, default is 1.

groupby_length [str, optional] Group time series and return n periods of length for each group.

padding [str, optional] Either Pandas frequency (e.g. ‘1D’) or ‘calendar_week’. If Pandas frequency, symmetric padding is undertaken, either side of length If ‘calendar_week’, padding is fit to the calendar week in which the extreme day(s) are found.

normalize [bool, optional] If True (default), data is normalized using normalized_copy().

kwargs [dict, optional] Dimensions of the selected var over which to index. Any remaining dimensions will be flattened by mean

calliope.core.time.masks.extreme_diff(data, tech0, tech1, var='resource', how='max', length='1D', n=1, groupby_length=None, padding=None, normalize=True, **kwargs)

Returns timesteps for period of length where the difference in extreme value for var between technologies tech0 and tech1 is either a minimum or a maximum.

Parameters

data [xarray.Dataset]

tech0 [str] First technology for which we find the extreme of var

tech1 [str] Second technology for which we find the extreme of var

var [str, optional] default 'resource'

how [str, optional] ‘max’ (default) or ‘min’.

length [str, optional] Defaults to ‘1D’.

n [int, optional] Number of periods of length to look for, default is 1.

groupby_length [str, optional] Group time series and return n periods of length for each group.

padding [str, optional] Either Pandas frequency (e.g. ‘1D’) or ‘calendar_week’. If Pandas frequency, symmetric padding is undertaken, either side of length If ‘calendar_week’, padding is fit to the calendar week in which the extreme day(s) are found.

normalize [bool, optional] If True (default), data is normalized using normalized_copy().

kwargs [dict, optional] Dimensions of the selected var over which to index. Any remaining dimensions will be flattened by mean

calliope.core.time.funcs.resample(data, timesteps, resolution)

Function to resample timeseries data from the input resolution (e.g. 1H), to the given resolution (e.g. 2H)

Parameters

data [xarray.Dataset] calliope model data, containing only timeseries data variables

timesteps [str or list; optional] If given, apply resampling to a subset of the timeseries data

resolution [str] time resolution of the output data, given in Pandas time frequency format. E.g. 1H = 1 hour, 1W = 1 week, 1M = 1 month, 1T = 1 minute. Multiples allowed.

2.1.3 Analyzing models

class calliope.analysis.plotting.plotting.ModelPlotMethods(model)

timeseries(**kwargs)
Parameters

array [str or list; default = ‘all’] options: ‘all’, ‘results’, ‘inputs’, the name/list of any energy carrier(s) (e.g. ‘power’), the name/list of any input/output DataArray(s).

User can specify ‘all’ for all input/results timeseries plots, ‘inputs’ for just input timeseries, ‘results’ for just results timeseries, or the name of any data array to plot (in either inputs or results). In all but the last case, arrays can be picked from dropdown in visualisation. In the last case, output can be saved to SVG and a rangeslider can be used.

timesteps_zoom [int, optional] Number of timesteps to show initially on the x-axis (if not given, the full time range is shown by default).

rangeslider [bool, optional] If True, displays a range slider underneath the plot for navigating (helpful primarily in interactive use).

subset [dict, optional] Dictionary by which data is subset (uses xarray loc indexing). Keys any of ['timeseries', 'locs', 'techs', 'carriers', 'costs'].

sum_dims [str, optional] List of dimension names to sum plot variable over.

squeeze [bool, optional] Whether to squeeze out dimensions of length = 1.

html_only [bool, optional; default = False] Returns a html string for embedding the plot in a webpage

to_file [False or str, optional; default = False] Will save plot to file with the given name and extension. to_file='plot.svg' to save to SVG, to_file='plot.png' for a static PNG image. Allowed file extensions are: ['png', 'jpeg', 'svg', 'webp'].

layout_updates [dict, optional] The given dict will be merged with the Plotly layout dict generated by the Calliope plotting function, overwriting keys that already exist.

plotly_kwarg_updates [dict, optional] The given dict will be merged with the Plotly plot function’s keyword arguments generated by the Calliope plotting function, overwriting keys that already exist.

capacity (**kwargs)

Parameters

array [str or list; default = ‘all’] options: ‘all’, ‘results’, ‘inputs’, the name/list of any energy capacity DataArray(s) from inputs/results. User can specify ‘all’ for all input/results capacities, ‘inputs’ for just input capacities, ‘results’ for just results capacities, or the name(s) of any data array(s) to plot (in either inputs or results). In all but the last case, arrays can be picked from dropdown in visualisation. In the last case, output can be saved to SVG.

orient [str, optional] ‘h’ for horizontal or ‘v’ for vertical barchart

subset [dict, optional] Dictionary by which data is selected (using xarray indexing loc[]). Keys any of ['timeseries', 'locs', 'techs', 'carriers', 'costs'].

sum_dims [str, optional] List of dimension names to sum plot variable over.

squeeze [bool, optional] Whether to squeeze out dimensions containing only single values.

html_only [bool, optional; default = False] Returns a html string for embedding the plot in a webpage

to_file [False or str, optional; default = False] Will save plot to file with the given name and extension. to_file='plot.svg' to save to SVG, to_file='plot.png' for a static PNG image. Allowed file extensions are: ['png', 'jpeg', 'svg', 'webp'].
layout_updates [dict, optional] The given dict will be merged with the Plotly layout dict generated by the Calliope plotting function, overwriting keys that already exist.

plotly_kwarg_updates [dict, optional] The given dict will be merged with the Plotly plot function’s keyword arguments generated by the Calliope plotting function, overwriting keys that already exist.

transmission(**kwargs)

Parameters

mapbox_access_token [str, optional] If given and a valid Mapbox API key, a Mapbox map is drawn for lat-lon coordinates, else (by default), a more simple built-in map.

html_only [bool, optional; default = False] Returns a html string for embedding the plot in a webpage

to_file [False or str, optional; default = False] Will save plot to file with the given name and extension. to_file=’plot.svg’ to save to SVG, to_file=’plot.png’ for a static PNG image. Allowed file extensions are: [‘png’, ‘jpeg’, ‘svg’, ‘webp’].

layout_updates [dict, optional] The given dict will be merged with the Plotly layout dict generated by the Calliope plotting function, overwriting keys that already exist.

plotly_kwarg_updates [dict, optional] The given dict will be merged with the Plotly plot function’s keyword arguments generated by the Calliope plotting function, overwriting keys that already exist.

summary(**kwargs)

Plot a summary containing timeseries, installed capacities, and transmission plots. Returns a HTML string by default, returns None if to_file given (and saves the HTML string to file).

Parameters

to_file [str, optional] Path to output file to save HTML to.

mapbox_access_token [str, optional] (passed to plot_transmission) If given and a valid Mapbox API key, a Mapbox map is drawn for lat-lon coordinates, else (by default), a more simple built-in map.

2.1.4 Pyomo backend interface

class calliope.backend.pyomo.interface.BackendInterfaceMethods(model)

access_model_inputs()

If the user wishes to inspect the parameter values used as inputs in the backend model, they can access a new Dataset of all the backend model inputs, including defaults applied where the user did not specify anything for a loc::tech

update_param(*args, **kwargs)

A Pyomo Param value can be updated without the user directly accessing the backend model.

Parameters

param [str] Name of the parameter to update

index [tuple of strings] Tuple of dimension indeces, in the order given in model.inputs for the reciprocal parameter

value [int, float, bool, or str] Value to assign to the Pyomo Param at the given index

Returns
Value will be updated in-place, requiring the user to run the model again to see the effect on results.

**activate_constraint** (*args, **kwargs)

Takes a constraint or objective name, finds it in the backend model and sets its status to either active or deactive.

**Parameters**

- **constraint** [str] Name of the constraint/objective to activate/deactivate Built-in constraints include ‘_constraint’
- **active** [bool, default=True] status to set the constraint/objective

**rerun** (*args, **kwargs)

Rerun the Pyomo backend, perhaps after updating a parameter value, (de)activating a constraint/objective or updating run options in the model model_data object (e.g. run.solver).

**Returns**

- **run_data** [xarray.Dataset] Raw data from this rerun, including both inputs and results. To filter inputs/results, use run_data.filter_by_attrs(is_result=...) with 0 for inputs and 1 for results.

### 2.1.5 Utility classes: AttrDict, Exceptions, Logging

**class** calliope.core.attrdict.AttrDict

A subclass of dict with key access by attributes:

```python
    d = AttrDict({'a': 1, 'b': 2})
    d.a == 1  # True
```

Includes a range of additional methods to read and write to YAML, and to deal with nested keys.

**copy()**

Override copy method so that it returns an AttrDict

**init_from_dict(d)**

Initialize a new AttrDict from the given dict. Handles any nested dicts by turning them into AttrDicts too:

```python
    d = AttrDict({'a': 1, 'b': {'x': 1, 'y': 2}})
    d.b.x == 1  # True
```

**classmethod from_yaml**

Returns an AttrDict initialized from the given path or file object f, which must point to a YAML file.

If resolve_imports is True, top-level import: statements are resolved recursively, else they are treated like any other key.

If resolve_imports is a string, such as foobar, import statements underneath that key are resolved, i.e. foobar.import:

When resolving import statements, anything defined locally overrides definitions in the imported file.

**classmethod from_yaml_string** (string, resolve_imports=True)

Returns an AttrDict initialized from the given string, which must be valid YAML.

**set_key** (key, value)

Set the given key to the given value. Handles nested keys, e.g.
```python
d = AttrDict()
d.set_key('foo.bar', 1)
d.foo.bar == 1  # True
```

**get_key** *(key, default=<_calliope.core.attrdict.__Missing object>)*
Looks up the given key. Like set_key(), deals with nested keys.

If default is anything but `_MISSING`, the given default is returned if the key does not exist.

**del_key** *(key)*
Delete the given key. Properly deals with nested keys.

**as_dict** *(flat=False)*
Return the AttrDict as a pure dict (with nested dicts if necessary).

**to_yaml** *(path=None)*
Saves the AttrDict to the path as a YAML file, or returns a YAML string if path is None.

**keys_nested** *(subkeys_as='list')*
Returns all keys in the AttrDict, sorted, including the keys of nested subdicts (which may be either regular dicts or AttrDicts).

If `subkeys_as='list'` (default), then a list of all keys is returned, in the form `['a', 'b.b1', 'b.b2']`.

If `subkeys_as='dict'`, a list containing keys and dicts of subkeys is returned, in the form `['a', {'b': ['bl', 'b2']}]`.

**union** *(other, allow_override=False, allow_replacement=False, allow_subdict_override_with_none=False)*
Merges the AttrDict in-place with the passed other AttrDict. Keys in other take precedence, and nested keys are properly handled.

If `allow_override` is False, a KeyError is raised if other tries to redefine an already defined key.

If `allow_replacement`, allow "_REPLACE_" key to replace an entire sub-dict.

If `allow_subdict_override_with_none` is False (default), a key of the form `this.that: None` in other will be ignored if subdicts exist in self like `this.that.foo: 1`, rather than wiping them.

**exception calliope.exceptions.ModelError**
ModelErrors should stop execution of the model, e.g. due to a problem with the model formulation or input data.

**exception calliope.exceptions.BackendError**

**exception calliope.exceptions.ModelWarning**
ModelWarnings should be raised for possible model errors, but where execution can still continue.

**exception calliope.exceptions.BackendWarning**
calliope.exceptions.print_warnings_and_raise_errors(warnings=None, errors=None)
Print warnings and raise ModelError from errors.

**Parameters**

- **warnings** [list, optional]
- **errors** [list, optional]

**calliope.core.util.logging.set_log_level(level)**
Set the minimum logging verbosity in a Python console. Higher verbosity levels will include their output and all those of following levels. Level options (in descending order of verbosity):
• ‘DEBUG’
• ‘SOLVER’ -> Calliope custom level, assigned value of 19, returns solver (e.g. GLPK) stream
• ‘INFO’ -> default level
• ‘WARNING’
• ‘ERROR’
• ‘CRITICAL’

2.2 Index
CHAPTER 3

Release history

3.1 Release History

3.1.1 0.6.3 (2018-10-03)

new Addition of flows plotting function. This shows production and how much they exchange with other locations. It also provides a slider in order to see flows’ evolution through time.

new calliope generate_runs in the command line interface can now produce scripts for remote clusters which require SLURM-based submission (sbatch...).

new backwards-incompatible Addition of scenarios, which complement and expand the existing overrides functionality. overrides becomes a top-level key in model configuration, instead of a separate file. The calliope run command has a new --scenario option which replaces --override_file, while calliope generate_runs has a new --scenarios option which replaces --override_file and takes a semicolon-separated list of scenario names or of group1,group2 combinations. To convert existing overrides to the new approach, simply group them under a top-level overrides key and import your existing overrides file from the main model configuration file with import: ['your_overrides_file.yaml'].

new Addition of calliope generate_scenarios command to allow building automating the construction of scenarios which consist of many combinations of overrides.

new Added --override_dict option to calliope run and calliope generate_runs commands

new Added solver performance comparison in the docs. CPLEX & Gurobi are, as expected, the best options. If going open-source & free, CBC is much quicker than GLPK!

new Calliope is tested and confirmed to run on Python 3.7

changed resource_unit - available to supply, supply_plus, and demand technologies - can now be defined as ‘energy_per_area’, ‘energy’, or ‘energy_per_cap’. ‘power’ has been removed. If ‘energy_per_area’ then available resource is the resource (CSV or static value) * resource_area, if ‘energy_per_cap’ it is resource * energy_cap. Default is ‘energy’, i.e. resource = available_resource.

changed Updated to xarray v0.10.8, including updates to timestep aggregation and NetCDF I/O to handle updated xarray functionality.
changed Removed calliope convert command. If you need to convert a 0.5.x model, first use calliope convert in Calliope 0.6.2 and then upgrade to 0.6.3 or higher.

changed Removed comment persistence in AttrDict and the associated API in order to improve compatibility with newer versions of ruamel.yaml

fixed Operate mode is more robust, by being explicit about timestep and loc_tech indexing in storage_initial preparation and resource_cap checks, respectively, instead of assuming an order.

fixed When setting ensure_feasibility, the resulting unmet_demand variable can also be negative, accounting for possible infeasibility when there is unused supply, once all demand has been met (assuming no load shedding abilities). This is particularly pertinent when the force_resource constraint is in place.

fixed When applying systemwide constraints to transmission technologies, they are no longer silently ignored. Instead, the constraint value is doubled (to account for the constant existence of a pair of technologies to describe one link) and applied to the relevant transmission techs.

fixed Permit groups in override files to specify imports of other YAML files

fixed If only interest_rate is defined within a cost class of a technology, the entire cost class is correctly removed after deleting the interest_rate key. This ensures an empty cost key doesn’t break things later on. Fixes issue #113.

fixed If time clustering with ‘storage_inter_cluster’ = True, but no storage technologies, the model doesn’t break. Fixes issue #142.

3.1.2 0.6.2 (2018-06-04)

new units_max_systemwide and units_equals_systemwide can be applied to an integer/binary constrained technology (capacity limited by units not energy_cap, or has an associated purchase (binary) cost). Constraint works similarly to existing energy_cap_max_systemwide, limiting the number of units of a technology that can be purchased across all locations in the model.

new backwards-incompatible primary_carrier for conversion_plus techs is now split into primary_carrier_in and primary_carrier_out. Previously, it only accounted for output costs, by separating it, om_con and om_prod are correctly accounted for. These are required conversion_plus essentials if there’s more than one input and output carrier, respectively.

new Storage can be set to cyclic using run.cyclic_storage. The last timestep in the series will then be used as the ‘previous day’ conditions for the first timestep in the series. This also applies to storage_inter_cluster, if clustering. Defaults to False, with intention of defaulting to True in 0.6.3.

new On clustering timeseries into representative days, an additional set of decision variables and constraints is generated. This addition allows for tracking stored energy between clusters, by considering storage between every datestep of the original (unclustered) timeseries as well as storage variation within a cluster.

new CLI now uses the IPython debugger rather than built-in pdb, which provides highlighting, tab completion, and other UI improvements

new AttrDict now persists comments when reading from and writing to YAML files, and gains an API to view, add and remove comments on keys

fixed Fix CLI error when running a model without transmission technologies

fixed Allow plotting for inputs-only models, single location models, and models without location coordinates

fixed Fixed negative om_con costs in conversion and conversion_plus technologies
3.1.3 0.6.1 (2018-05-04)

- New addition of user-defined datestep clustering, accessed by `clustering_func=file=filename.csv:column` in time aggregation config.
- New added `layout_updates` and `plotly_kwarg_updates` parameters to plotting functions to override the generated Plotly configuration and layout.
- Changed Cost class and sense (maximize/minimize) for objective function may now be specified in run configuration (default remains monetary cost minimization).
- Changed Cleaned up and documented `Model.save_commented_model_yaml()` method.
- Fixed Fixed error when calling `--save_plots` in CLI.
- Fixed Minor improvements to warnings.
- Fixed Pure dicts can be used to create a `Model` instance.
- Fixed `AttrDict.union` failed on all-empty nested dicts.

3.1.4 0.6.0 (2018-04-20)

Version 0.6.0 is an almost complete rewrite of most of Calliope’s internals. See user/whatsnew_060 for a more detailed description of the many changes.

**Major changes**

- Changed backwards-incompatible Substantial changes to model configuration format, including more verbose names for most settings, and removal of run configuration files.
- New backwards-incompatible Complete rewrite of Pyomo backend, including new various new and improved functionality to interact with a built model (see user/whatsnew_060).
- New Addition of a `calliope convert` CLI tool to convert 0.5.x models to 0.6.0.
- New Experimental ability to link to non-Pyomo backends.
- New New constraints: `resource_min_use` constraint for `supply` and `supply_plus` techs.
- Changed backwards-incompatible Removal of settings and constraints includes `subset_x`, `subset_y`, `s_time`, `r2`, `r_scale_to_peak`, `weight`.
- Changed backwards-incompatible `system_margin` constraint replaced with `reserve_margin` constraint.
- Changed backwards-incompatible Removed the ability to load additional custom constraints or objectives.

3.1.5 0.5.5 (2018-02-28)

- Fixed Allow `r_area` to be non-zero if either of `e_cap.max` or `e_cap.equals` is set, not just `e_cap.max`.
- Fixed Ensure static parameters in resampled timeseries are caught in constraint generation.
- Fixed Fix time masking when `set_t.csv` contains sub-hourly resolutions.
3.1.6 0.5.4 (2017-11-10)

**Major changes**

- fixed `r_area_per_e_cap` and `r_cap_equals_e_cap` constraints have been separated from `r_area` and `r_cap` constraints to ensure that user specified `r_area.max` and `r_cap.max` constraints are observed.

- changed technologies and location subsets are now communicated with the solver as a combined location:technology subset, to reduce the problem size, by ignoring technologies at locations in which they have not been allowed. This has shown drastic improvements in Pyomo preprocessing time and memory consumption for certain models.

**Other changes**

- fixed Allow plotting carrier production using `calliope.analysis.plot_carrier_production` if that carrier does not have an associated demand technology (previously would raise an exception).

- fixed Define time clustering method (sum/mean) for more constraints that can be time varying. Previously only included `r` and `e_eff`.

- changed storage technologies default `s_cap.max` to `inf`, not 0 and are automatically included in the `loc_tech_store` subset. This ensures relevant constraints are not ignored by storage technologies.

- changed Some values in the urban scale MILP example were updated to provide results that would show the functionality more clearly

- changed technologies have set colours in the urban scale example model, as random colours were often hideous.

- changed ruamel.yaml, not ruamel_yaml, is now used for parsing YAML files.

- fixed e_cap constraints for unmet_demand technologies are ignored in operational mode. Capacities are fixed for all other technologies, which previously raised an exception, as a fixed infinite capacity is not physically allowable.

- fixed stack_weights were strings rather than numeric datatypes on reading NetCDF solution files.

3.1.7 0.5.3 (2017-08-22)

**Major changes**

- new (BETA) Mixed integer linear programming (MILP) capabilities, when using `purchase` cost and/or `units.max/min/equals` constraints. Integer/Binary decision variables will be applied to the relevant technology-location sets, avoiding unnecessary complexity by describing all technologies with these decision variables.

**Other changes**

- changed YAML parser is now ruamel_yaml, not pyyaml. This allows scientific notation of numbers in YAML files (#57)

- fixed Description of PV technology in urban scale example model now more realistic

- fixed Optional ramping constraint no longer uses backward-incompatible definitions (#55)

- fixed One-way transmission no longer forces unidirectionality in the wrong direction
• fixed Edge case timeseries resource combinations, where infinite resource sneaks into an incompatible constraint, are now flagged with a warning and ignored in that constraint (#61)
• fixed e_cap.equals: 0 sets a technology to a capacity of zero, instead of ignoring the constraint (#63)
• fixed depreciation_getter now changes with location overrides, instead of just checking the technology level constraints (#64)
• fixed Time clustering now functions in models with time-varying costs (#66)
• changed Solution now includes time-varying costs (costs_variable)
• fixed Saving to NetCDF does not affect in-memory solution (#62)

3.1.8 0.5.2 (2017-06-16)
• changed Calliope now uses Python 3.6 by default. From Calliope 0.6.0 on, Python 3.6 will likely become the minimum required version.
• fixed Fixed a bug in distance calculation if both lat/lon metadata and distances for links were specified.
• fixed Fixed a bug in storage constraints when both s_cap and e_cap were constrained but no c_rate was given.
• fixed Fixed a bug in the system margin constraint.

3.1.9 0.5.1 (2017-06-14)
new backwards-incompatible Better coordinate definitions in metadata. Location coordinates are now specified by a dictionary with either lat/lon (for geographic coordinates) or x/y (for generic Cartesian coordinates), e.g. {lat: 40, lon: -2} or {x: 0, y: 1}. For geographic coordinates, the map_boundary definition for plotting was also updated in accordance. See the built-in example models for details.
new Unidirectional transmission links are now possible. See the documentation on transmission links.

Other changes
• fixed Missing urban-scale example model files are now included in the distribution
• fixed Edge cases in conversion_plus constraints addressed
• changed Documentation improvements

3.1.10 0.5.0 (2017-05-04)
Major changes
new Urban-scale example model, major revisions to the documentation to accommodate it, and a new calliope. examples module to hold multiple example models. In addition, the calliope new command now accepts a --template option to select a template other than the default national-scale example model, e.g.: calliope new my_urban_model --template=UrbanScale.
new Allow technologies to generate revenue (by specifying negative costs)
new Allow technologies to export their carrier directly to outside the system boundary
new Allow storage & supply_plus technologies to define a charge rate (c_rate), linking storage capacity (s_cap) with charge/discharge capacity (e_cap) by s_cap * c_rate => e_cap. As such, either s_cap.max & c_rate or e_cap.max &
c_rate can be defined for a technology. The smallest of $s_{cap}.max \times c_{rate}$ and $e_{cap}.max$ will be taken if all three are defined.

changed backwards-incompatible Revised technology definitions and internal definition of sets and subsets, in particular subsets of various technology types. Supply technologies are now split into two types: supply and supply_plus. Most of the more advanced functionality of the original supply technology is now contained in supply_plus, making it necessary to update model definitions accordingly. In addition to the existing conversion technology type, a new more complex conversion_plus was added.

Other changes

- changed backwards-incompatible Creating a Model() with no arguments now raises a ModelError rather than returning an instance of the built-in national-scale example model. Use the new calliope.examples module to access example models.
- changed Improvements to the national-scale example model and its tutorial notebook
- changed Removed SolutionModel class
- fixed Other minor fixes

3.1.11 0.4.1 (2017-01-12)

- new Allow profiling with the --profile and --profile_filename command-line options
- new Permit setting random seed with random_seed in the run configuration
- changed Updated installation documentation using conda-forge package
- fixed Other minor fixes

3.1.12 0.4.0 (2016-12-09)

Major changes

new Added new methods to deal with time resolution: clustering, resampling, and heuristic timestep selection

changed backwards-incompatible Major change to solution data structure. Model solution is now returned as a single xarray DataSet instead of multiple pandas DataFrames and Panels. Instead of as a generic HDF5 file, complete solutions can be saved as a NetCDF4 file via xarray’s NetCDF functionality.

While the recommended way to save and process model results is by NetCDF4, CSV saving functionality has now been upgraded for more flexibility. Each variable is saved as a separate CSV file with a single value column and as many index columns as required.

changed backwards-incompatible Model data structures simplified and based on xarray

Other changes

- new Functionality to post-process parallel runs into aggregated NetCDF files in calliope.read
- changed Pandas 0.18/0.19 compatibility
- changed 1.11 is now the minimum required numpy version. This version makes datetime64 tz-naive by default, thus preventing some odd behavior when displaying time series.
- changed Improved logging, status messages, and error reporting
• fixed Other minor fixes

3.1.13 0.3.7 (2016-03-10)

Major changes

changed Per-location configuration overrides improved. All technology constraints can now be set on a per-location basis, as can costs. This applies to the following settings:

• techname.x_map
• techname.constraints.*
• techname.constraints_per_distance.*
• techname.costs.*

The following settings cannot be overridden on a per-location basis:

• Any other options directly under techname, such as techname.parent or techname.carrier
• techname.costs_per_distance.*
• techname.depreciation.*

Other changes

• fixed Improved installation instructions
• fixed Pyomo 4.2 API compatibility
• fixed Other minor fixes

3.1.14 0.3.6 (2015-09-23)

• fixed Version 0.3.5 changes were not reflected in tutorial

3.1.15 0.3.5 (2015-09-18)

Major changes

new New constraint to constrain total (model-wide) installed capacity of a technology (e_cap.total_max), in addition to its per-node capacity (e_cap.max)

changed Removed the level option for locations. Level is now implicitly derived from the nested structure given by the within settings. Locations that define no or an empty within are implicitly at the topmost (0) level.

changed backwards-incompatible Revised configuration of capacity constraints: e_cap_max becomes e_cap.max, addition of e_cap.min and e_cap.equals (analogous for r_cap, s_cap, rb_cap, r_area). The e_cap.equals constraint supersedes e_cap_max_force (analogous for the other constraints). No backwards-compatibility is retained, models must change all constraints to the new formulation. See List of possible constraints for a complete list of all available constraints. Some additional constraints have name changes:

• e_cap_max_scale becomes e_cap_scale
• rb_cap_follows becomes rb_cap_follow, and addition of rb_cap_follow_mode
• s_time_max becomes s_time.max
changed backwards-incompatible All optional constraints are now grouped together, under constraints.
optional:

- \texttt{constraints.group\_fraction.group\_fraction} becomes \texttt{constraints.optional.group\_fraction}
- \texttt{constraints.ramping.ramping\_rate} becomes \texttt{constraints.optional.ramping\_rate}

Other changes

- new \texttt{analysis.map\_results} function to extract solution details from multiple parallel runs
- new Various other additions to analysis functionality, particularly in the \texttt{analysis\_utils} module
- new \texttt{analysis.get\_levelized\_cost} to get technology and location specific costs
- new Allow dynamically loading time mask functions
- changed Improved summary table in the model solution: now shows only aggregate information for transmission technologies, also added missing \texttt{s\_cap} column and technology type
- fixed Bug causing some total levelized transmission costs to be infinite instead of zero
- fixed Bug causing some CSV solution files to be empty

3.1.16 0.3.4 (2015-04-27)

- fixed Bug in construction and fixed O&M cost calculations in operational mode

3.1.17 0.3.3 (2015-04-03)

Major changes

changed In preparation for future enhancements, the ordering of location levels is flipped. The top-level locations at which balancing takes place is now level 0, and may contain level 1 locations. This is a backwards-incompatible change.

changed backwards-incompatible Refactored time resolution adjustment functionality. Can now give a list of masks in the run configuration which will all be applied, via \texttt{time.masks}, with a base resolution via \texttt{time.resolution} (or instead, as before, load a resolution series from file via \texttt{time.file}). Renamed the \texttt{time\_functions} submodule to \texttt{time\_masks}.

Other changes

- new Models and runs can have a \texttt{name}
- changed More verbose \texttt{calliope run}
- changed Analysis tools restructured
- changed Renamed \texttt{debug.keepfiles} setting to \texttt{debug.keep\_temp\_files} and better documented debug configuration
3.1.18 0.3.2 (2015-02-13)

- new Run setting `model_override` allows specifying the path to a YAML file with overrides for the model configuration, applied at model initialization (path is given relative to the run configuration file used). This is in addition to the existing `override` setting, and is applied first (so `override` can override `model_override`).

- new Run settings `output.save_constraints` and `output.save_constraints_options`

- new Run setting `parallel.post_run`

- changed Solution column names more in line with model component names

- changed Can specify more than one output format as a list, e.g. `output.format: ['csv', 'hdf']`

- changed Run setting `parallel.additional_lines renamed to parallel.pre_run`

- changed Better error messages and CLI error handling

- fixed Bug on saving YAML files with numpy dtypes fixed

- Other minor improvements and fixes

3.1.19 0.3.1 (2015-01-06)

- Fixes to time_functions

- Other minor improvements and fixes

3.1.20 0.3.0 (2014-12-12)

- Python 3 and Pyomo 4 are now minimum requirements

- Significantly improved documentation

- Improved model solution management by saving to HDF5 instead of CSV

- Calculate shares of technologies, including the ability to define groups for the purpose of computing shares

- Improved operational mode

- Simplified time_tools

- Improved output plotting, including dispatch, transmission flows, and installed capacities, and added model configuration to support these plots

- `r` can be specified as power or energy

- Improved solution speed

- Better error messages and basic logging

- Better sanity checking and error messages for common mistakes

- Basic distance-dependent constraints (only implemented for e_loss and cost of e_cap for now)

- Other improvements and fixes
3.1.21 0.2.0 (2014-03-18)

- Added cost classes with a new set $k$
- Added energy carriers with a new set $c$
- Added conversion technologies
- Speed improvements and simplifications
- Ability to arbitrarily nest model configuration files with `import` statements
- Added additional constraints
- Improved configuration handling
- Ability to define timestep options in run configuration
- Cleared up terminology (nodes vs locations)
- Improved TimeSummarizer masking and added new masks
- Removed technology classes
- Improved operational mode with results output matching planning mode and dynamic updating of parameters in model instance
- Working parallel_tools
- Improved documentation
- Apache 2.0 licensed
- Other improvements and fixes

3.1.22 0.1.0 (2013-12-10)

- Some semblance of documentation
- Usable built-in example model
- Improved and working TimeSummarizer
- More flexible masking for TimeSummarizer
- Ability to add additional constraints without editing core source code
- Some basic test coverage
- Working parallel run configuration system
Copyright 2013-2018 Calliope contributors listed in AUTHORS

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.


C

calliope, 1

calliope.backend.pyomo.constraints.capacity,
  79
calliope.backend.pyomo.constraints.conversion,
  86
calliope.backend.pyomo.constraints.conversion_plus,
  86
calliope.backend.pyomo.constraints.costs,
  82
calliope.backend.pyomo.constraints.dispatch,
  80
calliope.backend.pyomo.constraints.energy_balance,
  76
calliope.backend.pyomo.constraints.export,
  83
calliope.backend.pyomo.constraints.milp,
  84
calliope.backend.pyomo.constraints.network,
  87
calliope.backend.pyomo.constraints.policy,
  87
calliope.backend.pyomo.objective, 76
calliope.backend.pyomo.variables, 75
calliope.core.time.clustering, 92
calliope.core.timefuncs, 93
calliope.core.timemasks, 92
calliope.core.util.logging, 97
calliope.examples, 47
calliope.exceptions, 97
Index

A
access_model_inputs() (calliope.backend.pyomo.interface.BackendInterfaceMethods method), 95
activate_constraint() (calliope.backend.pyomo.interface.BackendInterfaceMethods method), 96
as_dict() (calliope.core.attrdict.AttrDict method), 97
AttrDict (class in calliope.core.attrdict), 96

B
BackendError, 97
BackendInterfaceMethods (class in calliope.backend.pyomo.interface), 95
BackendWarning, 97
balance_conversion_constraint_rule() (in module calliope.backend.pyomo.constraints.conversion), 86
balance_conversion_plus_tiers_constraint_rule() (in module calliope.backend.pyomo.constraints.conversion_plus), 87
balance_demand_constraint_rule() (in module calliope.backend.pyomo.constraints.energy_balance), 77
balance_supply_constraint_rule() (in module calliope.backend.pyomo.constraints.energy_balance), 76
balance_transmission_constraint_rule() (in module calliope.backend.pyomo.constraints.energy_balance), 78
balance_supply_plus_constraint_rule() (in module calliope.backend.pyomo.constraints.energy_balance), 78

calliope (module), 1
calliope.backend.pyomo.constraints.capacity (module), 79
calliope.backend.pyomo.constraints.conversion (module), 86
calliope.backend.pyomo.constraints.conversion_plus (module), 86
calliope.backend.pyomo.constraints.costs (module), 82
calliope.backend.pyomo.constraints.dispatch (module), 80
calliope.backend.pyomo.constraints.energy_balance (module), 76
calliope.backend.pyomo.constraints.export (module), 83
calliope.backend.pyomo.constraints.milp (module), 84
calliope.backend.pyomo.constraints.network (module), 87
calliope.backend.pyomo.constraints.policy (module), 87
calliope.backend.pyomo.objective (module), 76
calliope.backend.pyomo.variables (module), 75
calliope.core.time.clustering (module), 92
calliope.core.time.funcs (module), 93
calliope.core.time.masks (module), 92
calliope.core.util.logging (module), 97
calliope.examples (module), 47
calliope.exceptions (module), 97
capacity() (calliope.analysis.plotting.plotting.ModelPlotMethods method), 94
carrier_consumption_max_constraint_rule() (in module calliope.backend.pyomo.constraints.dispatch), 81
carrier_consumption_max_milp_constraint_rule() (in module calliope.backend.pyomo.constraints.milp), 85
<table>
<thead>
<tr>
<th>Letter</th>
<th>Function/Method Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td><code>national_scale()</code> (in module calliope.examples), 47</td>
</tr>
<tr>
<td>O</td>
<td><code>operate()</code> (in module calliope.examples), 47</td>
</tr>
<tr>
<td>P</td>
<td><code>print_warnings_and_raise_errors()</code> (in module calliope.exceptions), 97</td>
</tr>
<tr>
<td>R</td>
<td><code>ramping_constraint()</code> (in module calliope.backend.pyomo.constraints.dispatch), 81</td>
</tr>
<tr>
<td></td>
<td><code>ramping_down_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 81</td>
</tr>
<tr>
<td></td>
<td><code>ramping_up_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 81</td>
</tr>
<tr>
<td></td>
<td><code>rerun()</code> (calliope.backend.pyomo.interface.BackendInterfaceMethods method), 96</td>
</tr>
<tr>
<td></td>
<td><code>resample()</code> (in module calliope.core.time.funcs), 93</td>
</tr>
<tr>
<td></td>
<td><code>reserve_margin_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.policy), 88</td>
</tr>
<tr>
<td></td>
<td><code>resource_area_capacity_per_loc_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.capacity), 80</td>
</tr>
<tr>
<td></td>
<td><code>resource_area_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.capacity), 79</td>
</tr>
<tr>
<td></td>
<td><code>resource_area_per_energy_capacity_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.capacity), 79</td>
</tr>
<tr>
<td></td>
<td><code>resource_availability_supply_plus_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.energy_balance), 77</td>
</tr>
<tr>
<td></td>
<td><code>resource_capacity_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.capacity), 79</td>
</tr>
<tr>
<td></td>
<td><code>resource_capacity equals_energy_capacity_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.capacity), 79</td>
</tr>
<tr>
<td></td>
<td><code>resource_max_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 81</td>
</tr>
<tr>
<td></td>
<td><code>run()</code> (calliope.Model method), 91</td>
</tr>
<tr>
<td>S</td>
<td><code>save_commented_model_yaml()</code> (calliope.Model method), 91</td>
</tr>
<tr>
<td></td>
<td><code>set_key()</code> (calliope.core.attrdict.AttrDict method), 96</td>
</tr>
<tr>
<td></td>
<td><code>set_log_level()</code> (in module calliope.core.util.logging), 97</td>
</tr>
<tr>
<td></td>
<td><code>storage_capacity_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.capacity), 79</td>
</tr>
<tr>
<td></td>
<td><code>storage_capacity_max_purchase_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.milp), 85</td>
</tr>
<tr>
<td></td>
<td><code>storage_capacity_min_purchase_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.milp), 85</td>
</tr>
<tr>
<td></td>
<td><code>storage_capacity_units_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.milp), 85</td>
</tr>
<tr>
<td></td>
<td><code>storage_initial_rule()</code> (in module calliope.backend.pyomo.constraints.energy_balance), 78</td>
</tr>
<tr>
<td></td>
<td><code>storage_inter_max_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 82</td>
</tr>
<tr>
<td></td>
<td><code>storage_inter_min_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 82</td>
</tr>
<tr>
<td></td>
<td><code>storage_intra_max_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 82</td>
</tr>
<tr>
<td></td>
<td><code>storage_intra_min_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 82</td>
</tr>
<tr>
<td></td>
<td><code>storage_max_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.dispatch), 81</td>
</tr>
<tr>
<td></td>
<td><code>summary()</code> (calliope.analysis.plotting.plotting.ModelPlotMethods method), 95</td>
</tr>
<tr>
<td></td>
<td><code>symmetric_transmission_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.network), 87</td>
</tr>
<tr>
<td></td>
<td><code>system_balance_constraint_rule()</code> (in module calliope.backend.pyomo.constraints.energy_balance), 76</td>
</tr>
<tr>
<td>T</td>
<td><code>time_clustering()</code> (in module calliope.examples), 47</td>
</tr>
<tr>
<td></td>
<td><code>time_masking()</code> (in module calliope.examples), 48</td>
</tr>
<tr>
<td></td>
<td><code>time_resampling()</code> (in module calliope.examples), 48</td>
</tr>
<tr>
<td></td>
<td><code>timeseries()</code> (calliope.analysis.plotting.plotting.ModelPlotMethods method), 93</td>
</tr>
<tr>
<td></td>
<td><code>to_csv()</code> (calliope.Model method), 91</td>
</tr>
<tr>
<td></td>
<td><code>to_netcdf()</code> (calliope.Model method), 91</td>
</tr>
<tr>
<td></td>
<td><code>to_yaml()</code> (calliope.core.attrdict.AttrDict method), 97</td>
</tr>
</tbody>
</table>
transmission() (calliope.analysis.plotting.plotting.ModelPlotMethods method), 95

U
union() (calliope.core.attrdict.AttrDict method), 97
unit_capacity_constraint_rule() (in module calliope.backend.pyomo.constraints.milp), 84
unit_capacity_systemwide_constraint_rule() (in module calliope.backend.pyomo.constraints.milp), 86
unit_commitment_constraint_rule() (in module calliope.backend.pyomo.constraints.milp), 84
update_costs_investment_purchase_constraint() (in module calliope.backend.pyomo.constraints.milp), 86
update_costs_investment_units_constraint() (in module calliope.backend.pyomo.constraints.milp), 85
update_costs_var_constraint() (in module calliope.backend.pyomo.constraints.export), 83
update_param() (calliope.backend.pyomo.interface.BackendInterfaceMethods method), 95
update_system_balance_constraint() (in module calliope.backend.pyomo.constraints.export), 83
urban_scale() (in module calliope.examples), 47