Calliope Documentation

Release 0.6.1

Calliope contributors

May 04, 2018

Contents

v0.6.1 (*Release history*)

This is the documentation for version 0.6.1. 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).

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

CHAPTER 1

User guide

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 of 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]*.

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.

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 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.

1.2.4 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 New in v0.6.0

Version *0.6* is backwards incompatible with version *0.5*. If you are familiar with how Calliope functions then this page will act as a reference for moving to version *0.6*.

1.3.1 Converting a 0.5.x model to 0.6.0

Version *0.6* provides a conversion script to models created with Calliope 0.5 into 0.6-compatible models:

```
calliope convert run.yaml model.yaml output_dir
```

`run.yaml` and `model.yaml` are the main run and model configuration files of the 0.5 model to be converted, while `output_dir` is a directory into which the converted model will be saved.

The file structure of the input model is preserved (all YAML and CSV input files and their folder structure), but all information from the run configuration file is merged into the main model configuration file.

Time series are handled by the script: the `set_t.csv` required in Calliope 0.5 models is removed, with the time step information inserted directly into each of the remaining time series CSV files.

The conversion script has some important limitations:

- Comments in YAML files are lost
- Parallel run configurations are not converted to the new override group configuration format
- The `group_fraction` constraint is not converted, as its formulation has changed substantially in 0.6.0
- `carrier_ratios` are not converted
- Operational mode configuration is not converted

Functionality which is not converted, has been removed in version 0.6.0, or keys not known to the script, are moved into a `__disabled` key in the output YAML files, so they remain visible for the user to remove or to manually convert if needed.

The script also adds `interest_rate` and `lifetime` for each technology, using the implicit default values from Calliope 0.5 (25 years lifetime, a 0.10 interest rate for the monetary cost class and 0 for other cost classes).

See also:

0.6.0 model configuration changes

This page lists the full contents of `calliope/config/conversion_0.6.0.yaml`, which documents the changes in model configuration from 0.5.x to 0.6.0:

```
# Structure of this file:
# Each major section of model / run configuration has its own top-level key.
# Within each top-level section, changes are given as ``old name: new name``,
# and if the specific setting has been removed, the new name is ``null``,
# possibly with a comment indicating

run_config:
    import: null  # use import statements in model configuration file instead
    subset_y: null
    subset_x: null
    subset_t: model.subset_time
    solver: run.solver
    solver_options: run.solver_options
    name: null
    model: null  # since there is no separate run configuration file any more, there␣
↪is no need to specify a model configuration file
    mode: run.mode
    output.format: null
    output.path: null
    debug.keep_temp_files: run.save_logs
    debug.symbolic_solver_labels: null
    override: null  # now achieved with override groups, see override.yaml in examples
    parallel: null
    random_seed: model.random_seed
    debug: null  # no longer available
```

(continues on next page)

(continued from previous page)

```
model_config:
    opmode: null
    startup_time: null
    name: model.name
    data_path: model.timeseries_data_path
    objective: null  # custom objective not implemented yet
    constraints: null  # custom constraints not implemented yet
    system_margin: model.reserve_margin
    group_fraction: null  # major change to structure of this constraint, not␣
↪automatically converted
    metadata: null  ### SPECIAL LOGIC: move coordinates over to location configuration


tech_config:
    parent: essentials.parent
    group: null  # Use tech_groups to specify groups
    name: essentials.name
    stack_weight: null  # stack_weight is no longer supported
    color: essentials.color
    x_map: null  # now achieved by directly specifying file=filename.csv:column
    carrier: essentials.carrier
    primary_carrier: essentials.primary_carrier
    carrier_in: essentials.carrier_in # If conversion_plus, now a list of carrier␣
↪names. Ratios between carriers found in constraints.carrier_ratios
    carrier_in_2: essentials.carrier_in_2 # If conversion_plus, now a list of carrier␣
↪names. Ratios between carriers found in constraints.carrier_ratios
    carrier_in_3: essentials.carrier_in_3 # If conversion_plus, now a list of carrier␣
↪names. Ratios between carriers found in constraints.carrier_ratios
    carrier_out: essentials.carrier_out # If conversion_plus, now a list of carrier␣
↪names. Ratios between carriers found in constraints.carrier_ratios
    carrier_out_2: essentials.carrier_out_2 # If conversion_plus, now a list of␣
↪carrier names. Ratios between carriers found in constraints.carrier_ratios
    carrier_out_3: essentials.carrier_out_3 # If conversion_plus, now a list of␣
↪carrier names. Ratios between carriers found in constraints.carrier_ratios
    export: constraints.export_carrier
    constraints.r: constraints.resource
    constraints.force_r: constraints.force_resource
    constraints.r_unit: constraints.resource_unit
    constraints.r_eff: constraints.resource_eff
    constraints.r_area.min: constraints.resource_area_min
    constraints.r_area.max: constraints.resource_area_max
    constraints.r_area.equals: constraints.resource_area_equals
    constraints.r_area_per_e_cap: constraints.resource_area_per_energy_cap
    constraints.r_cap.min: constraints.resource_cap_min
    constraints.r_cap.max: constraints.resource_cap_max
    constraints.r_cap.equals: constraints.resource_cap_equals
    constraints.r_cap_equals_e_cap: constraints.resource_cap_equals_energy_cap
    constraints.r_scale: constraints.resource_scale
    constraints.r_scale_to_peak: constraints.resource_scale_to_peak
    constraints.s_init: constraints.storage_initial
    constraints.s_cap.min: constraints.storage_cap_min
    constraints.s_cap.max: constraints.storage_cap_max
    constraints.s_cap.equals: constraints.storage_cap_equals
    constraints.s_cap_per_unit: constraints.storage_cap_per_unit
    constraints.c_rate: constraints.charge_rate
```

(continues on next page)

(continued from previous page)

```
    constraints.s_time.max: null
    constraints.use_s_time: null
    constraints.s_loss: constraints.storage_loss
    constraints.e_prod: constraints.energy_prod
    constraints.e_con: constraints.energy_con
    constraints.p_eff: constraints.parasitic_eff
    constraints.e_eff: constraints.energy_eff
    constraints.e_eff_per_distance: constraints.energy_eff_per_distance
    constraints.e_cap.min: constraints.energy_cap_min
    constraints.e_cap.max: constraints.energy_cap_max
    constraints.e_cap.equals: constraints.energy_cap_equals
    constraints.e_cap_total.max: constraints.energy_cap_max_systemwide
    constraints.e_cap_total.equals: constraints.energy_cap_equals_systemwide
    constraints.e_cap_scale: constraints.energy_cap_scale
    constraints.e_cap.min_use: constraints.energy_cap_min_use
    constraints.e_cap_min_use: constraints.energy_cap_min_use
    constraints.e_cap_per_unit: constraints.energy_cap_per_unit
    constraints.e_ramping: constraints.energy_ramping
    constraints.export_cap: constraints.export_cap
    constraints.export_carrier: constraints.export_carrier
    constraints.units.min: constraints.units_min
    constraints.units.max: constraints.units_max
    constraints.units.equals: constraints.units_equals
    constraints.r_scale_to_peak: null
    constraints.allow_r2: null
    constraints.r2_startup_only: null
    constraints.r2_eff: null
    constraints.r2_cap.min: null
    constraints.r2_cap.max: null
    constraints.r2_cap.equals: null
    constraints.r2_cap_follow: null
    constraints.r2_cap_follow_mode: null
    constraints.s_time.max: null
    weight: null
    per_distance: null


tech_constraints_per_distance_config:
    e_loss: constraints.energy_eff_per_distance


tech_costs_config:
    s_cap: storage_cap
    r_area: resource_area
    r_cap: resource_cap
    r2_cap: null
    e_cap: energy_cap
    om_frac: om_annual_investment_fraction
    om_fixed: om_annual
    om_var: om_prod
    om_fuel: om_con
    export: export
    purchase: purchase


tech_costs_per_distance_config:
    e_cap: energy_cap_per_distance
```

(continues on next page)

(continued from previous page)

```
location_config:
    override: techs
    techs: null # List as keys of the subdict 'techs'
    within: null


###


depreciation_config:  # manually processed in convert.py, listed here for completeness
    plant_life: constraints.lifetime
    interest: costs.{cost_class}.interest_rate
```

1.3.2 Removed functionality

If you require any of the removed functionality, we recommend you open an issue on GitHub for it to be built into a later revision of *0.6*.

Technology constraints

- *s_time* (providing a minimum/maximum/exact time of stored energy available for discharge) no longer exists. This constraint was relatively unpredictable in its effects when providing any combination of *s_cap*, *e_cap*, *c_rate* and time clustering.
- The variable *r2* (providing a secondary resource that could be used by a supply/supply_plus technology), along with all its constraints, have been removed. To utilise multiple resource inputs, conversion_plus can be used instead.
- *r_scale_to_peak* (allowing a user to provide a value for the peak resource to which the entire time series would be scaled accordingly) has been removed. *resource_scale* (previously *r_scale*) can still be used for scaling resource values by the given scale factor.
- *weight* (giving a technology a disproportionate weight in the objective function calculation) has been removed.

Custom objectives

The ability to load additional constraints or objectives has been removed. It is still possible to define a custom objective, but to load it, a modeller needs to use a development installation of Calliope and load the function manually.

See also:

Development guide

1.3.3 Updated functionality

Verbosity

Almost all sets, constraints, costs, and variables have been updated to be more verbose, making models more readable. The primary updates are:

Sets

- *y* -> *techs*
- *x* -> *locs*
- *c* -> *carriers*
- *k* -> *costs*

Constraints/Costs

- *e* -> *energy*, e.g. *e_cap* -> *energy_cap*
- *r* -> *resource*, e.g. *r_cap* -> *resource_cap*
- *s* -> *storage*, e.g. *s_cap* -> *storage_cap*
- *c_rate* -> *charge_rate*
- *p_eff* -> *parasitic_eff*

Variables

- *r* -> *resource_con*: an output from the model giving how much of a resource was consumed
- *r* -> *resource*: the available resource as an input parameter to the model
- *c_prod/c_con* -> *carrier_prod/carrier_con*: The produced/consumed carrier energy in each time storage_cap

Model and run configuration

run.yaml no longer exists. Instead, all information needed to run a model is now stored in *model.yaml* under the headings *model* and *run*.

run only contains information about the solver: which one to use and any specific solver options to apply.

model contains all other information: time subsetting, model mode, output format, parallel runs, and time clustering.

To solve a model, point to the *model.yaml* file, e.g.: `calliope run path/to/model.yaml`.

Overrides

Overrides are no longer applied within *run.yaml* (or even *model.yaml*). Instead, overrides are grouped and placed into a separate YAML file, called for example *overrides.yaml*.

Each group defines any number of overrides to the technology, location, link, model, or run definitions. One or several such groups can then be applied when solving a model, e.g.:

overrides.yaml:

```
higher_costs:
    techs.ccgt.costs.monetary.energy_cap: 10
    locations.region2.techs.csp.costs.monetary.energy_cap: 100
winter:
    model.subset_time: ['2005-01-01', '2005-02-28']
```

Running in the command line:

```
calliope run model.yaml --override_file=overrides.yaml:higher_costs

calliope run model.yaml --override_file=overrides.yaml:higher_costs,winter
```

Running interactively:

```
# only apply the 'higher_costs' override group
model = calliope.Model(
    'model.yaml',
    override_file='overrides.yaml:higher_costs'
)

# apply both the 'higher_costs' and 'winter' override groups
model2 = calliope.Model(
    'model.yaml',
    override_file='overrides.yaml:higher_costs,winter'
)
```

As in version *0.5*, overrides can be applied when creating a *Model* object, via the argument *override_dict*. A dictionary can then be given:

```
higher_costs = {
    'techs.ccgt.costs.monetary.energy_cap': 10,
    'locations.region2.techs.csp.costs.monetary.energy_cap': 100
}

model = calliope.Model('model.yaml', override_dict=higher_costs)
```

Parallel runs

Building on the simplified way to define overrides (see above) and on lessons learnt during the development of Calliope so far, the functionality to generate multiple runs to run either on a single machine or in parallel on a high-performance cluster has been greatly simplified and improved.

See also:

Generating scripts to run a model many times

Location and technology subsets

In model configuration, *subset_x* and *subset_y* (subsetting the used locations and technologies, respectively) no longer exist. *subset_t*, now called *subset_time*, does still exist.

To remove specific technologies or locations from a model, the new and much more powerful `exists` option can be used.

See also:

Removing techs, locations and links

Technology definition

A technology is now defined in three parts: *essentials*, *constraints*, and *costs*. All top-level definitions (*parent*, *carrier_out*, etc.) are now given under *essentials* and cannot be defined per-location – they are defined only once for a

given technology and apply model-wide. Both *constraints* and *costs* remain the same as in *0.5*, but with more verbose naming:

Old:

```
supply_grid_power:
    name: 'National grid import'
    parent: supply
    carrier: power
    constraints:
        r: inf
        e_cap.max: 2000
    costs:
        monetary:
            e_cap: 15
            om_fuel: 0.1
```

New:

```
supply_grid_power:
    essentials:
        name: 'National grid import'
        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
```

Carrier ratios and export carriers have also been moved from essentials into constraints:

Old:

```
chp:
    name: 'Combined heat and power'
    stack_weight: 100
    parent: conversion_plus
    export: true
    primary_carrier: power
    carrier_in: gas
    carrier_out: power
    carrier_out_2:
        heat: 0.8
    constraints:
        e_cap.max: 1500
        e_eff: 0.405
    costs:
        monetary:
            e_cap: 750
            om_var: 0.004
            export: file=export_power.csv
```

New:

```
chp:
    essentials:
        name: 'Combined heat and power'
        parent: conversion_plus
        primary_carrier: 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
            export: file=export_power.csv
```

Per distance constraints and costs have now been incorporated under the constraints and costs keys, with a '_per_distance' suffix:

Old:

```
heat_pipes:
    name: 'District heat distribution'
    parent: transmission
    carrier: heat
    constraints:
        e_cap.max: 2000
    constraints_per_distance:
        e_loss: 0.025
    costs_per_distance:
        monetary:
            e_cap: 0.3
```

New:

```
heat_pipes:
    essentials:
        name: 'District heat distribution'
        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
```

Interest rates and life times

As seen in the above examples, technology lifetime and interest rate must now be defined for each technology, under *costs*. In version *0.5*, technologies not defining these would silently use implicit default values of 0.10 for interest rate and 25 years for life time. Setting these explicitly for any technology which has investment costs (i.e. those which are not *om_...* or *export*) is now mandatory; no default values exist any more.

Location definition

In version *0.5*, location definitions included a list of technologies to permit at that location(s). An additional *overrides* key permitted per-location changes to model-wide technology definitions.

In *0.6*, "overriding" refers only to model-wide overrides applied *as described above*. At each location, *techs* simply lists all allowed technologies and any possible changes to model-wide configuration values to apply at this location only, as shown below:

Old:

```
locations:
    region1:
        techs: [ccgt, csp]
        overrides:
            ccgt:
                constraints:
                    energy_cap: 100
```

New:

```
locations:
    region1:
        techs:
            ccgt:
                constraints:
                    energy_cap: 100
            # Note that csp must be listed to be permitted here,
            # even though it has no location-specific configuration.
            csp:
```

Loading time series data from CSV files

x_map (mapping a technology name to a column in a CSV file) has been removed. Instead, a user can define the time series file column when defining the file name, separated from the file name by a *:*. If no column name is provided, Calliope will look for a column with the location name.

Old:

```
# will look for the column `demand` in the file `demand_heat_r.csv`
locations:
    region1:
        techs: [demand_power]
            overrides:
                demand_power:
                    x_map: demand
                    constraints:
                        r: file
```

New:

```
# will look for the column `demand` in the file `demand_heat_r.csv`
locations:
    region1:
        techs:
            demand_power:
                constraints:
                    resource: file=demand_heat.csv:demand
```

Link definition

Links have remained much the same as before. However, there is a slightly different structure in defining technologies, bringing the definition of link technologies more in line with the rest of the model configuration format.

Old:

```
links:
    region1,region2:
        ac_transmission:
            constraints:
                e_cap: 1000
```

New:

```
links:
    region1,region2:
        techs:
            ac_transmission:
                constraints:
                    energy_cap: 1000
```

Location metadata

Location coordinates, previously given under the *metadata* key, are now given directly per location:

Old:

```
metadata:
    # metadata given in cartesian coordinates, not lat, lon.
    map_boundary:
        lower_left:
            x: 0
            y: 0
        upper_right:
            x: 1
            y: 1
    location_coordinates:
        region1: {x: 2, y: 7}
        region2: {x: 8, y: 7}
```

New:

```
locations:
    region1:
```

(continues on next page)

(continued from previous page)

```
        techs:
            ccgt:
            csp:
        coordinates: {x: 2, y: 7}
    region2:
        techs:
            demand_power:
        coordinates: {x: 8, y: 7}
```

`group_share` constraint

The `group_fraction` constraint is now called `group_share` and has a different formulation more in line with the rest of the tech-specific constraints:

```
group_share:
    csp,ccgt:
        energy_cap_min: 0.5
        energy_cap_max: 0.9
        carrier_prod_min:
            power: 0.5
```

In the process of making these updates, the `demand_power_peak` and (undocumented) `ignored_techs` options were removed from `group_share`.

`charge_rate`

When first introduced, charge rate was used to hard-link *energy_cap* and *storage_cap* for a storage/supply_plus technology. This meant that on defining `energy_cap_max` and `charge_rate`, a user was implicitly defining `storage_cap_max`. This hard-link has now been removed, replaced with only one constraint concerning charge rate: $storage_{cap}(loc :: tech) \geq energy_{cap}(loc : tech) \times charge\_rate(loc : tech)$.

See also:

Capacity

Pre-processed data

Version *0.5* kept pre-processed data in either a dictionary (static data), pandas dataframe (location data) or an xarray Dataset (timeseries data). To view a value that would be used in optimisation, the user would call *model.get_option()*. Similarly, to edit a value before running the model, a user could use *model.set_option()*.

Now, all pre-processed data is held in a single unified xarray Dataset: *model.inputs*.

To view and edit this data before it is sent to the solver, a user need only use standard xarray functionality (see their documentation for more information).

Plotting data

Note: Advanced plotting is still under construction. In case our current functionality is insufficient, input and output data can be plotted by the user using their preferred Python plotting tools, or any other language that can access either NetCDF or CSV data.

Plotting functions can now be called directly on the model and now use Plotly instead of *0.5*'s matplotlib.

Changes are:

- `calliope.analysis.plot_capacity(model.solution)` to `model.plot.capacity()`
- `calliope.analysis.plot_transmission(model.solution, carrier='power', tech='ac_transmission')` to `model.plot.transmission()`
- `calliope.analysis.plot_carrier_production(model.solution, carrier='power')` to `model.plot.timeseries()`

All available data is plotted, with dropdown menus available for a user to move between plots. A summary of all plotting can also be produced using `model.plot.summary()`, a function that is also available via the command line interface.

See also:

Model class

Operational mode

In *0.6*, running in operational mode changes capacities from decision variables to parameters, preventing various issues that plagued operational mode in prior versions. Additional sense checks were added to ensure that functionality incompatible with operational mode, such as time clustering, is not accidentally used together with it.

See also:

Operational mode

1.3.4 New functionality

Debugging & checks

A user can now output a data structure of all model input data (the *model_run* dictionary) after Calliope's internal pre-processing, into a YAML file, for debugging. This debug file includes comments as to where constraint/cost values have originated (e.g. having been set by a location-specific configuration, or from a model-wide override group).

Similarly, sense checks are undertaken at several points during pre-processing to ensure the model being built is robust. This includes checks for missing data, possibly misspelled constraints, incompatible inputs, and much more.

This functionality will not find all possible user input errors, as this is an impossible task. However, it flags common mistakes, and the format of implementation allows for further checks to be applied in the future.

Pre-processed model

Having the pre-processed model available in one xarray Dataset allows a model to be saved to file *before* being run. Although pre-processing is quick, this allows a user to avoid pre-processing the same file multiple times. Instead, they can read in a previously saved NetCDF file which fully describes the model.

Multiple backends

Our primary solver backend is Pyomo. However, we have now extracted all pre-processing stages from the backend, with all data for a model run being stored in a single xarray Dataset. This permits the implementation of additional backends.

One such backend currently in an experimental state is based on JuMP in the Julia programming language. Linking Calliope to Julia is a long-term project, for which we welcome any contributions.

Pyomo warmstart

Warmstart functionality can be used in solvers other than GLPK. They allow a previously constructed model to be changed slightly without having to be fully rebuilt. This can speed up re-running a model when you have just a few input parameters you would like to change (the cost of a technology, for instance).

Although the use of warmstart existed in operational mode in version *0.5*, now it extends to all possible parameters in all models. This functionality is currently undocumented in Calliope, but the Pyomo documentation provides some information and the Pyomo model built by Calliope can be accessed by *model._backend_model*.

Backend interface

Once the backend model has been built, it can be accessed by a user, via Calliope. Parameters can be checked and changed, constraints can be activated/deactivated and a model can be re run, all without having to build the backend again. User who are familiar with building large models with Pyomo will be aware of the time penalty associated with processing the model in Pyomo. This additional functionality helps mitigate this, as changing a few parameters need not require complete model rebuild.

See also:

Pyomo backend interface

Logging

In an interactive Python session (e.g. using Jupyter notebook), output from Calliope can be triggered at different levels of verbosity. By default on building the model (`calliope.Model()`) and running it (`model.run()`), there is no logging displayed unless it is at least a *WARNING*. For helpful information on where the model is in its pre-processing and running in the solver, verbosity can be increased using `calliope.set_log_level()`.

See also:

Utility classes: AttrDict, Exceptions, Logging

1.4 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*.

1.4.1 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

1.4.2 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
    - overrides.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:

```
$ 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*

1.4.3 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`:

```
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*

1.4.4 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:

```
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:

```
ccgt:
    essentials:
        name: 'Combined cycle gas turbine'
        color: '#FDC97D'
        parent: supply
        carrier_out: power
    constraints:
        resource: inf
        energy_eff: 0.5
```

(continues on next page)

(continued from previous page)

```
        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:

```
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.

1.4.5 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:

```
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 constraints. If given, location-specific constraints 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`:

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

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 such as costs. As in the case of locations, the `links` section can specify per-link constraints that supersede any model-wide constraints.

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*.

1.4.6 Run configuration (`run`)

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

```
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:

```
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*.

1.4.7 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 "override groups" in a separate file (in the above example, `overrides.yaml`):

```
run1:
    model.subset_time: ['2005-01-01', '2005-01-31']
run2:
    model.subset_time: ['2005-02-01', '2005-02-31']
```

Each group is given by a name (above, `run1` and `run2`) and any number of model settings – anything in the model configuration can be overridden by an override group. In the above example, the two runs specify different time subsets, so would run an otherwise identical model over two different periods of time series data.

One or several override groups can be applied when running a model, as described in *Running a model*. They can also be used to generate scripts that run a Calliope model with slightly changed settings many times, either sequentially, or in parallel on a high-performance cluster.

See also:

Generating scripts to run a model many times

1.5 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.

1.5.1 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.
- `--override_file={filename.yaml}:{override_groups}`: Specify override groups to apply to the model (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.

Overrides

Assuming we have specified an override group called `milp` in a file called `overrides.yaml`, we can apply it to our model with:

```
$ calliope run testmodel/model.yaml --override_file=overrides.yaml:milp --save_
↪netcdf=results.nc
```

Multiple overrides from the YAML file can be applied at once. For example, we may want to change some of the costs through an additional override group called `high_cost_scenario`. We could then use `--override_file=overrides.yaml:milp,high_cost_scenario` to apply both overrides simultaneously.

See also:

Analysing a model, *Overrides*

1.5.2 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:

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

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).

Overrides

There are two ways to apply override groups interactively:

1. By setting the *override_file* argument analogously to use in the command-line tool, e.g.:

```
model = calliope.Model(
    'model.yaml',
    override_file='overrides.yaml:milp'
)
```

2. By passing the *override_dict* argument, which is a Python dictionary or `AttrDict` of overrides:

```
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:

```
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.

1.5.3 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*.

1.5.4 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 acheived.

See also:

Time resolution adjustment, Stefan Pfenninger (2017). Dealing with multiple decades of hourly wind and PV time series in energy models: a comparison of methods to reduce time resolution and the planning implications of inter-annual variability. Applied Energy.

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.

See also:

Solver options

Rerunning a model

After running, if there is an infeasibility you want to address, or simply a few values you dont 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 you current model.results and rebuild he 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

1.5.5 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*.

1.6 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.

1.6.1 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.

1.6.2 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.

Summary plots

If you want all the data in one place, you can run `model.plot.summary(out_file='path/to/file.html')`, which will build a HTML file of all the interactive plots (maintaining the interactivity) and save it to `out_file`. This HTML file can be opened in a web browser to show all the plots. This funcionality is made avaiable 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.

1.6.3 Reading solutions

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

```
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.

1.7 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)

1.7.1 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 `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 (`energy_cap`) and a constraint on its maximum built capacity.

Fig. 2: The layout of a supply node, in this case `ccgt`, which has an infinite resource, a carrier conversion efficiency ($energy_{eff}$), and a constraint on its maximum built $energy_{cap}$ (which puts an upper limit on $energy_{prod}$).

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

```
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
```

There are a few things to note. First, `ccgt` defines essential information: a name, a color (given as an HTML color code, for later visualisation), its parent, `supply`, and its 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).

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 `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 `csp` (concentrating solar power), and serves as an example of a complex 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
        energy_eff: 0.4
```

(continues on next page)

(continued from previous page)

```
        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 contraints 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 *resevoir*). 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:

Fig. 5: A simple demand node.

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

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):

Fig. 6: A simple transmission node with an $energy_{eff}$.

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

`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:

Fig. 7: Locations and their technologies in the example model

Let's now look at the first location definition:

```
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
```

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.

1.7.2 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 'inifinitely' 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 ($energy_{eff}$) 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:

```
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
```

The final supply technology is `pv` (solar photovoltaic power), which serves as an inflexible supply technology. It has a time-dependant resource availablity, loaded from file, a maximum area over which it can capture its resource (`resource_area_max`) and a requirement that all available resource must be used (`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 `parasitic_eff` and the `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 `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:

```
pv:
    essentials:
        name: 'Solar photovoltaic power'
        color: '#F9D956'
        parent: supply_power_plus
    constraints:
        export_carrier: electricity
        resource: file=pv_resource.csv  # 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
```

(continues on next page)

(continued from previous page)

```
        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 `supply_plus`, but rather `supply_power_plus`. We use this to show the possibility of an intermediate technology group, which provides the information on the energy carrier (`electricity`) and the ultimate abstract base technology (`supply_plus`):

```
tech_groups:
    supply_power_plus:
        essentials:
            parent: supply_plus
            carrier: electricity
```

Intermediate technology groups allow us to avoid repetition of technology information, be it in `essentials`, `constraints`, or `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:

```
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
```

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:

```
chp:
    essentials:
        name: 'Combined heat and power'
        color: '#E4AB97'
        parent: conversion_plus
        primary_carrier: 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
```

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 eletricity. 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'
```

(continues on next page)

(continued from previous page)

```
        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:

Fig. 13: Locations and their technologies in the urban-scale example model

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:
            constraints.resource: file=demand_heat.csv
    available_area: 500
    coordinates: {x: 2, y: 7}
```

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:
    techs:
        boiler:
            costs.monetary.energy_cap: 43.1 # different boiler costs
        pv:
            costs.monetary:
```

(continues on next page)

(continued from previous page)

```
                    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:
            constraints.resource: file=demand_power.csv
        demand_heat:
            constraints.resource: file=demand_heat.csv
    available_area: 900
    coordinates: {x: 5, y: 3}
```

`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.

1.7.3 Tutorial 3: Mixed Integer Linear Programming

This example is based on the *urban scale example model*, but with an override. An override file exists in which binary and integer decision variables are triggered, creating a MILP model, rather than the conventional Calliope LP model.

> **Warning:** Integer and Binary variables are still experimental and may not cover all edge cases as intended. Please raise an issue on GitHub if you see unexpected behaviour.

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.

1.8 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.

1.8.1 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
```

(continues on next page)

(continued from previous page)

```
        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:

```
links:
    location1,location2:
        transmission-tech:
            constraints:
                one_way: true
```

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
```

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 multipyling 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 consisten 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.

Fig. 14: Representation of the `supply_plus` technology

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.

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.

Fig. 15: Representation of the most complex `conversion_plus` technology available

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: 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: 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

```
cchp:
    essentials:
        name: Combined cooling, heat and power
        carrier_in: gas
        carrier_out: electricity
        carrier_out_2: [heat, cooling]
        primary_carrier: electricity

    constraints:
        energy_eff: 0.45
        energy_cap_max: 100
        carrier_ratios.carrier_out_2: {heat: 0.8, cooling: 0.5}
```

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).

```
gt:
    essentials:
        name: Advanced gas turbine
        carrier_in: [methane, hydrogen]
        carrier_out: electricity

    constraints:
        energy_eff: 0.5
        energy_cap_max: 100
        carrier_ratios:
            carrier_in: {methane: 1, hydrogen: 0.5}
```

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*.

```
complex:
    essentials:
        name: Complex fictional technology
        carrier_in: [coal, gas, oil]
        carrier_in_2: [biomass, waste]
        carrier_out: [high_T_heat, electricity]
        carrier_out_2: [mid_T_heat, cooling]
        carrier_out_3: low_T_heat
        primary_carrier: electricity

    constraints:
        energy_eff: 1
        energy_cap_max: 100
        carrier_ratios:
            carrier_in: {coal: 1.2, gas: 1, oil: 1.6}
            carrier_in_2: {biomass: 1, waste: 1.25}
            carrier_out: {high_T_heat: 0.8, electricity: 0.6}
            carrier_out_2: {mid_T_heat: 0.3, cooling: 0.2}
            carrier_out_3.low_T_heat: 0.15
```

A `primary_carrier` must be defined when there are multiple `carrier_out` values defined. `primary_carrier` can be defined as any carrier in a technology's output carriers (including secondary and tertiary carriers). The chosen carrier will be the one to which costs are applied.

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 an 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:

```
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

techs:
    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
```

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:

```
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:

```
model:
    group_share:
        csp,cold_fusion:
            energy_cap_max: 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 override groups in the *built-in national-scale example*'s `overrides.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 through override groups, model components can be removed for debugging or scenario analysis.

This works for:

- Techs: `techs.tech_name.exists: false`
- Locations: `locations.location_name.exists: false`
- Links: `links.location1,location2.exists: false`
- Techs at a specific location: `locations.location_name.techs.tech_name.exists: false`
- Transmission techs at a specific location: `links.location1,location2.techs.transmission_tech.exists: false`

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 be 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

Override groups 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` command-line tool 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, and `bsub` creates a submission script for a bsub-based high-performance cluster.
- `--override_file`: The file that specifies override groups.
- `--groups`: A semicolon-separated list of override groups to generate scripts for, for example, `run1;run2`. A comma is used to group override groups together into a single model – for example, `run1,high_costs;run1,low_costs` would run the model twice, once applying the `run1` and `high_costs` override groups, and once applying `run1` and `low_costs`.

A fully-formed command generating a Windows batch script to run a model four times with each of the override groups "run1", "run2", "run3", and "run4":

```
calliope generate_runs model.yaml run_model.bat --kind=windows --override_
↪file=overrides.yaml --groups "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 --override_file=overrides.yaml --groups "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}_{groups}.nc` (model results) and `plots_{run_number}_{groups}.html` (HTML plots), where `{run_number}` is the run number and `{groups}` is the specified set of groups. On a cluster, log files are saved to files with names starting with `log_` in the same directory.

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 a 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
```

1.8.2 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
- overrides.yaml
+ timeseries_data
    - csp_resource.csv
    - demand-1.csv
    - demand-2.csv
+ model_config
    - locations.yaml
    - techs.yaml
```

model.yaml:

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

model:
    name: National-scale example model

    # What version of Calliope this model is intended for
    calliope_version: 0.6.1

    # Time series data path - can either be a path relative to this file, or an␣
↪absolute path
```

(continues on next page)

(continued from previous page)

```
    timeseries_data_path: 'timeseries_data'

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

run:
    solver: glpk

    ensure_feasibility: true # Switching on unmet demand

    bigM: 1e6 # setting 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

    mode: plan  # Choices: plan, operate
```

overrides.yaml:

```
##
# Overrides for different example model configuratiions
##

profiling:
    model.name: 'National-scale example model (profiling run)'
    model.subset_time: ['2005-01-01', '2005-01-15']
    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_mode:
    run.mode: operate
    run.operation:
            window: 12
            horizon: 24
    techs.csp.constraints.charge_rate: 0.0162857697 # energy_cap_max / storage_cap_max
    techs.csp.constraints.energy_cap_max: null
    techs.battery.constraints.storage_cap_max: null

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

(continues on next page)

(continued from previous page)

```
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:
    model.subset_time: ['2005-01-01', '2005-01-31']
run2:
    model.subset_time: ['2005-02-01', '2005-02-31']
run3:
    model.subset_time: ['2005-01-01', '2005-01-31']
    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
            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 together
            csp,cold_fusion:
                carrier_prod_min:
                    power: 0.85

group_share_cold_fusion_cap:
    model:
```

(continues on next page)

(continued from previous page)

```
        group_share:
            # At most 20% of total energy_cap can come from CSP and cold fusion␣
↪together
            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

techs:

    ##
    # Supply
    ##

    # ccgt-start
    ccgt:
```

(continues on next page)

(continued from previous page)

```
        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
            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
    # csp-end

    ##
    # Storage
    ##
    # battery-start
    battery:
        essentials:
            name: 'Battery storage'
            color: '#3B61E3'
            parent: storage
            carrier: power
        constraints:
```

(continues on next page)

(continued from previous page)

```
            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
        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

##
# TRANSMISSION CAPACITIES
##

links:
    # 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:

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

model:
    name: Urban-scale example model

    # What version of Calliope this model is intended for
    calliope_version: 0.6.1

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

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

run:
    mode: plan  # Choices: plan, operate

    solver: glpk

    ensure_feasibility: true # Switching on unmet demand

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

overrides.yaml:

```
##
# Overrides for different example model configuratiions
##

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:
```

(continues on next page)

(continued from previous page)

```
            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.istance: 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
                supply_grid_power.constraints.energy_cap_max: 40
                supply_gas.constraints.energy_cap_max: 700

        X2:
            techs:
                boiler.constraints.energy_cap_max: 200
                pv.constraints.energy_cap_max: 70
                supply_gas.constraints.energy_cap_max: 250

        X3:
            techs:
                boiler.constraints.energy_cap_max: 0
                pv.constraints.energy_cap_max: 50
                supply_gas.constraints.energy_cap_max: 0

    links:
        X1,X2.techs.power_lines.constraints.energy_cap_max: 300
        X1,X3.techs.power_lines.constraints.energy_cap_max: 60
        X1,N1.techs.heat_pipes.constraints.energy_cap_max: 300
        N1,X2.techs.heat_pipes.constraints.energy_cap_max: 250
        N1,X3.techs.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:
```

(continues on next page)

(continued from previous page)

```
        masks:
            - {function: extreme_diff, options: {tech0: demand_heat, tech1: demand_
↪electricity, how: max, n: 2}}
        function: resample
        function_options: {resolution: 6H}
```

techs.yaml:

```
##
# TECHNOLOGY DEFINITIONS
##

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

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

techs:

##-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
```

(continues on next page)

(continued from previous page)

```
    # supply-end

##-Renewables-##
    # pv-start
    pv:
        essentials:
            name: 'Solar photovoltaic power'
            color: '#F9D956'
            parent: supply_power_plus
        constraints:
            export_carrier: electricity
            resource: file=pv_resource.csv  # 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
            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
    # boiler-start
    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
    # boiler-end

# Conversion_plus
    # chp-start
    chp:
        essentials:
            name: 'Combined heat and power'
            color: '#E4AB97'
            parent: conversion_plus
            primary_carrier: electricity
            carrier_in: gas
            carrier_out: electricity
            carrier_out_2: heat
```

(continues on next page)

(continued from previous page)

```
        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-end

##-DEMAND-##
    # demand-start
    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
```

(continues on next page)

(continued from previous page)

```
            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:
                constraints.resource: file=demand_power.csv
            demand_heat:
                constraints.resource: file=demand_heat.csv
        available_area: 900
```

(continues on next page)

(continued from previous page)

```
        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
```

1.8.3 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

Setting	Default	Comments
name		Model name
calliope_version		Calliope framework version this model is intended for
timeseries_data_path		Path to time series data
timeseries_dateformat	%Y-%m-%d %H:%M:%S	Timestamp format of all time series data when read from file
timeseries_data		Dict of dataframes with time series data (when passing in dicts rather than YAML files to Model constructor)
subset_time		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'
reserve_margin	{}	Per-carrier system-wide reserve margins
random_seed		Seed for random number generator used during clustering
time	{}	Optional settings to adjust time resolution, see *Time resolution adjustment* for the available options
group_share	{}	Optional settings for the group_share constraint

List of run settings

Setting	Default	Comments
backend	pyomo	Backend to use to build and solve the model. As of v0.6.0, only *pyomo* is available
solver	glpk	Which solver to use
solver_options		A list of options, which are passed on to the chosen solver, and are therefore solver-dependent
solver_io		What method the Pyomo backend should use to communicate with the solver
save_logs		Directory into which to save logs and temporary files. Also turns on symbolic solver labels in the Pyomo backend
bigM	1000000000.0	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
ensure_feasibility	False	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
operation	{}	Settings for operational mode
objective	minmax_cost_optimization	Name of internal objective function to use, currently only min/max cost-based optimisation is available
objective_options	{}	Arguments to pass to objective function. If cost-based objective function in use, should include 'cost_class' and 'sense' (maximize/minimize)
mode	plan	Which mode to run the model in: 'plan' or 'operation'
zero_threshold	1e-10	Any value coming out of the backend that is smaller than this threshold (due to floating point errors, probably) will be set to zero

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`.

Setting	Default	Name	Unit	Comments
lifetime		Technology lifetime	years	Must be defined if fixed capital costs are defined. A reasonable value for many technologies is around 20-25 years.
carrier_ratios	{}	Carrier ratios	fraction	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.
resource	0	Available resource	kWh/m^2 | kW/m^2	Maximum available resource (static, or from file as timeseries). Unit dictated by `reosurce_unit`
force_resource	False	Force resource	boolean	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
resource_unit	power	Resource unit	N/A	Sets the unit of `resource` to either `power` (i.e. kW) or `energy` (i.e. kWh), which affects how resource time series are processed when performing time resolution adjustments
resource_eff	1.0	Resource efficiency	fraction	Efficiency (static, or from file as timeseries) in capturing resource before it reaches storage (if storage is present) or conversion to carrier.
resource_area_min	0	Minimum installed collector area	m^2	
resource_area_max	False	Maximum installed collector area	m^2	Set to false by default in order to disable this constraint
resource_area_equals	False	Specific installed collector area	m^2	
resource_area_per_energy_cap	False	Energy capacity per unit collector ares	boolean	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`)
resource_cap_min	0	Minimum installed resource consumption capacity	kW	
resource_cap_max	inf	Maximum installed resource consumption capacity	kW	

Continued on next page

Table 1 – continued from previous page

Setting	Default	Name	Unit	Comments
resource_cap_equals	False	Specific installed resource consumption capacity	kW	overrides `_max` and `_min` constraints.
resource_cap_equals_energy_cap	False	Resource capacity equals energy cpacity	boolean	If true, `resource_cap` is forced to equal `energy_cap`
resource_min_use	False	Minimum resource consumption	fraction	Set to a value between 0 and 1 to force minimum resource consumption for production technologies
resource_scale	1.0	Resource scale	fraction	Scale resource (either static value or all valuesin timeseries) by this value
storage_initial	0	Initial storage level	kWh	Set stored energy in device at the first timestep
storage_cap_min	0	Minimum storage capacity	kWh	
storage_cap_max	inf	Maximum storage capacity	kWh	If not defined, `energy_cap_max * charge_rate` will be used as the capacity.
storage_cap_equals	False	Specific storage capacity	kWh	If not defined, `energy_cap_equals * charge_rate` will be used as the capacity and overrides `_max` and `_min` constraints.
storage_cap_per_unit	False	Storage capacity per purchased unit	kWh/unit	Set the storage capacity of each integer unit of a technology perchased.
charge_rate	False	Charge rate	hour $^{-1}$	ratio of maximum charge/discharge (kW) for a given maximum storage capacity (kWh)
storage_loss	0	Storage loss rate	hour $^{-1}$	rate of storage loss per hour (static, or from file as timeseries), used to calculate lost stored energy as `(1 - storage_loss)^hours_per_timestep`
energy_prod	False	Energy production	boolean	Allow this technology to supply energy to the carrier (static boolean, or from file as timeseries).
energy_con	False	Energy consumption	boolean	Allow this technology to consume energy from the carrier (static boolean, or from file as timeseries).
parasitic_eff	1.0	Plant parasitic efficiency	fraction	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
energy_eff	1.0	Energy efficiency	fraction	conversion efficiency (static, or from file as timeseries), from `resource`/`storage`/`carrier_in` (tech dependent) to `carrier_out`.
energy_eff_per_distance	1.0	Energy efficiency per distance	distance $^{-1}$	Set as value between 1 (no loss) and 0 (all energy lost)

Continued on next page

Table 1 – continued from previous page

Setting	Default	Name	Unit	Comments
energy_cap_min	0	Minimum installed energy capacity	kW	Limits decision variables `carrier_prod`/`carrier_con` to a minimum/maximum.
energy_cap_max	inf	Maximum installed energy capacity	kW	Limits decision variables `carrier_prod`/`carrier_con` to a maximum/minimum.
energy_cap_equals	False	Specific installed energy capacity	kW	fixes maximum/minimum if decision variables `carrier_prod`/`carrier_con` and overrides `_max` and `_min` constraints.
energy_cap_max_systemwide	inf	System-wide maximum installed energy capacity	kW	Limits the sum to a maximum/minimum, for a particular technology, of the decision variables `carrier_prod`/`carrier_con` over all locations.
energy_cap_equals_systemwide	False	System-wide specific installed energy capacity	kW	fixes the sum to a maximum/minimum, for a particular technology, of the decision variables `carrier_prod`/`carrier_con` over all locations.
energy_cap_scale	1.0	Energy capacity scale	float	Scale all `energy_cap` min/max/equals/total_max/total_equals constraints by this value
energy_cap_min_use	False	Minimum carrier production	fraction	Set to a value between 0 and 1 to force minimum carrer 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.
energy_cap_per_unit	False	Energy capacity per purchased unit	kW/unit	Set the capacity of each integer unit of a technology purchased
energy_ramping	False	Ramping rate	fraction / hour	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.
export_cap	inf	Export capacity	kW	Maximum allowed export of produced energy carrier for a technology.
export_carrier		Export carrier	N/A	Name of carrier to be exported. Must be an output carrier of the technology
units_min	False	Minimum number of purchased units	integer	Turns the model from LP to MILP.
units_max	False	Maximum number of purchased units	integer	Turns the model from LP to MILP.

Continued on next page

Table 1 – continued from previous page

Setting	Default	Name	Unit	Comments
units_equals	False	Specific number of purchased units	integer	Turns the model from LP to MILP.

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`.

Setting	Default	Name	Unit	Comments
interest_rate	0	Interest rate	fraction	Used when computing levelized costs
storage_cap	0	Cost of storage capacity	kWh $^{-1}$	
resource_area	0	Cost of resource collector area	m^{-2}	
resource_cap	0	Cost of resource consumption capacity	kW $^{-1}$	
energy_cap	0	Cost of energy capacity	kW $_{gross}^{-1}$	
energy_cap_per_distance	0	Cost of energy capacity, per unit distance	kW $_{gross}^{-1}$ / distance	Applied to transmission links only
om_annual_investment_fraction	0	Fractional yearly O&M costs	fraction / total investment	
om_annual	0	Yearly O&M costs	kW $_{energy\_cap}^{-1}$	
om_prod	0	Carrier production cost	kWh $^{-1}$	Applied to carrier production of a technology
om_con	0	Carrier consumption cost	kWh $^{-1}$	Applied to carrier consumption of a technology
export	0	Carrier export cost	kWh $^{-1}$	Usually used in the negative sense, as a subsidy.
purchase	0	Purchase cost	unit $^{-1}$	Triggers a binary variable for that technology to say that it has been purchased or is applied to integer variable `units`

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:

```
essentials:
  parent:
costs: {}
constraints:
  resource: inf
  resource_unit: power
  energy_prod: true
```

Required constraints, allowed constraints, and allowed costs:

```
allowed_constraints:
-   energy_prod
-   lifetime
-   resource
-   force_resource
-   resource_min_use
-   resource_unit
-   resource_area_min
-   resource_area_max
-   resource_area_equals
-   resource_area_per_energy_cap
-   resource_scale
-   energy_eff
-   energy_cap_min
-   energy_cap_max
-   energy_cap_equals
-   energy_cap_max_systemwide
-   energy_cap_equals_systemwide
-   energy_cap_scale
-   energy_cap_min_use
-   energy_cap_per_unit
-   energy_ramping
-   energy_eff_per_distance
-   export_cap
```

(continues on next page)

(continued from previous page)

```
-   export_carrier
-   units_min
-   units_max
-   units_equals
required_constraints:
- [energy_cap_max, energy_cap_equals, energy_cap_per_unit]
allowed_costs:
-   interest_rate
-   resource_area
-   energy_cap
-   om_annual_investment_fraction
-   om_annual
-   om_prod
-   om_con
-   export
-   purchase
-   depreciation_rate
```

supply_plus

Default constraints provided by the parent tech group:

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

Required constraints, allowed constraints, and allowed costs:

```
allowed_constraints:
-   energy_prod
-   lifetime
-   resource
-   force_resource
-   resource_min_use
-   resource_unit
-   resource_eff
-   resource_area_min
-   resource_area_max
-   resource_area_equals
-   resource_area_per_energy_cap
-   resource_cap_min
-   resource_cap_max
-   resource_cap_equals
-   resource_cap_equals_energy_cap
-   resource_scale
-   parasitic_eff
-   energy_eff
-   energy_cap_min
```

(continues on next page)

(continued from previous page)

```
-   energy_cap_max
-   energy_cap_equals
-   energy_cap_max_systemwide
-   energy_cap_equals_systemwide
-   energy_cap_scale
-   energy_cap_min_use
-   energy_cap_per_unit
-   energy_ramping
-   energy_eff_per_distance
-   export_cap
-   export_carrier
-   units_min
-   units_max
-   units_equals
-   storage_initial
-   storage_cap_min
-   storage_cap_max
-   storage_cap_equals
-   storage_cap_per_unit
-   charge_rate
-   storage_time_max
-   storage_loss
required_constraints:
- [energy_cap_max, energy_cap_equals, energy_cap_per_unit]
allowed_costs:
-   interest_rate
-   storage_cap
-   resource_area
-   resource_cap
-   energy_cap
-   om_annual_investment_fraction
-   om_annual
-   om_prod
-   om_con
-   export
-   purchase
-   depreciation_rate
```

demand

Default constraints provided by the parent tech group:

```
essentials:
  parent:
costs: {}
constraints:
  resource_unit: power
  force_resource: true
  energy_con: true
```

Required constraints, allowed constraints, and allowed costs:

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

storage

Default constraints provided by the parent tech group:

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

Required constraints, allowed constraints, and allowed costs:

```
allowed_constraints:
-   energy_prod
-   energy_con
-   lifetime
-   energy_eff
-   energy_cap_min
-   energy_cap_max
-   energy_cap_equals
-   energy_cap_max_systemwide
-   energy_cap_equals_systemwide
-   energy_cap_scale
-   energy_cap_min_use
-   energy_cap_per_unit
-   energy_ramping
-   storage_initial
-   storage_cap_min
-   storage_cap_max
-   storage_cap_equals
-   storage_cap_per_unit
-   charge_rate
-   storage_time_max
-   storage_loss
-   export_cap
-   export_carrier
-   units_min
-   units_max
-   units_equals
required_constraints:
- [energy_cap_max, energy_cap_equals, energy_cap_per_unit]
```

(continues on next page)

(continued from previous page)

```
- [storage_cap_max, storage_cap_equals]
allowed_costs:
-   interest_rate
-   storage_cap
-   energy_cap
-   om_annual_investment_fraction
-   om_annual
-   om_prod
-   export
-   purchase
-   depreciation_rate
```

transmission

Default constraints provided by the parent tech group:

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

Required constraints, allowed constraints, and allowed costs:

```
allowed_constraints:
-   energy_prod
-   energy_con
-   lifetime
-   energy_con
-   energy_prod
-   energy_eff_per_distance
-   energy_eff
-   one_way
-   energy_cap_scale
required_constraints:
- [energy_cap_max, energy_cap_equals, energy_cap_per_unit]
allowed_costs:
-   interest_rate
-   energy_cap
-   energy_cap_per_distance
-   om_annual_investment_fraction
-   om_annual
-   om_prod
-   purchase
-   purchase_per_distance
-   depreciation_rate
```

conversion

Default constraints provided by the parent tech group:

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

Required constraints, allowed constraints, and allowed costs:

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

conversion_plus

Default constraints provided by the parent tech group:

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

Required constraints, allowed constraints, and allowed costs:

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

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:

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

can be written as:

```
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:

```
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 *AttrDict*s, 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.

1.8.4 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

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

Defines decision variables.

Variable	Dimensions
energy_cap	loc_techs
carrier_prod	loc_tech_carriers_prod, timesteps
carrier_con	loc_tech_carriers_con, timesteps
cost	costs, loc_techs_cost
resource_area	loc_techs_area,
storage_cap	loc_techs_store
storage	loc_techs_store, timesteps
resource_con	loc_techs_supply_plus, timesteps
resource_cap	loc_techs_supply_plus
carrier_export	loc_tech_carriers_export, timesteps
cost_var	costs, loc_techs_om_cost, timesteps
cost_investment	costs, loc_techs_investment_cost
purchased	loc_techs_purchase
units	loc_techs_milp
operating_units	loc_techs_milp, timesteps
unmet_demand	loc_carriers, timesteps

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.

$$min: z = \sum_{loc::tech_{cost}} cost(loc :: tech, cost = cost_k)) + \sum_{loc::carrier,timestep} unmet\_demand(loc :: carrier, timestep) \times bigM$$

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_mod*, *loc_carrier*, *timestep*)

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

$$\sum_{loc::tech::carrier_{prod} \in loc::carrier} \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) + \sum_{loc::tech::carrier_{con} \in loc::carrier} \boldsymbol{carrier_{con}}(loc$$

calliope.backend.pyomo.constraints.energy_balance.**balance_supply_constraint_rule**(*backend_mod*, *loc_tech*, *timestep*)

Limit production from supply techs to their available resource

$$min\_use(loc :: tech) \times available\_resource(loc :: tech, timestep) \leq \frac{\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep)}{\eta_{energy}(loc :: tech, timestep)} \geq avai$$

If $force\_resource(loc :: tech)$ is set:

$$\frac{\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep)}{\eta_{energy}(loc :: tech, timestep)} = available\_resource(loc :: tech, timestep) \quad \forall loc :: tech \in loc :: techs_{supp}$$

Where:

$$available\_resource(loc :: tech, timestep) = resource(loc :: tech, timestep) \times resource\_scale(loc :: tech)$$

if $loc :: tech$ is in $loc :: techs_{area}$:

$$available\_resource(loc :: tech, timestep) = resource(loc :: tech, timestep) \times resource\_scale(loc :: tech) \times \boldsymbol{resource_{are}}$$

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

Limit consumption from demand techs to their required resource.

$$\boldsymbol{carrier_{con}}(loc :: tech :: carrier, timestep) \times \eta_{energy}(loc :: tech, timestep) \geq required\_resource(loc :: tech, timestep)$$

If $force\_resource(loc :: tech)$ is set:

$$\boldsymbol{carrier_{con}}(loc :: tech :: carrier, timestep) \times \eta_{energy}(loc :: tech, timestep) = required\_resource(loc :: tech, timestep)$$

Where:

$required\_resource(loc :: tech, timestep) = resource(loc :: tech, timestep) \times resource\_scale(loc :: tech)$

if $loc :: tech$ is in $loc :: techs_{area}$:

$required\_resource(loc :: tech, timestep) = resource(loc :: tech, timestep) \times resource\_scale(loc :: tech) \times \boldsymbol{resource_{are}}$

calliope.backend.pyomo.constraints.energy_balance.**resource_availability_supply_plus_constra**

Limit production from supply_plus techs to their available resource.

$\boldsymbol{resource_{con}}(loc :: tech, timestep) \leq available\_resource(loc :: tech, timestep) \quad \forall loc :: tech \in loc :: techs_{supply^+}, \forall time$

If $force\_resource(loc :: tech)$ is set:

$\boldsymbol{resource_{con}}(loc :: tech, timestep) = available\_resource(loc :: tech, timestep) \quad \forall loc :: tech \in loc :: techs_{supply^+}, \forall time$

Where:

$available\_resource(loc :: tech, timestep) = resource(loc :: tech, timestep) \times resource_{scale}(loc :: tech)$

if $loc :: tech$ is in $loc :: techs_{area}$:

$available\_resource(loc :: tech, timestep) = resource(loc :: tech, timestep) \times resource_{scale}(loc :: tech) \times resource_{area}(lo$

calliope.backend.pyomo.constraints.energy_balance.**balance_transmission_constraint_rule**(*backe*
loc_t
times

Balance carrier production and consumption of transmission technologies

$-1 * \boldsymbol{carrier_{con}}(loc_{from} :: tech : loc_{to} :: carrier, timestep) \times \eta_{energy}(loc :: tech, timestep) = \boldsymbol{carrier_{prod}}(loc_{to} :: tech$

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

calliope.backend.pyomo.constraints.energy_balance.**balance_supply_plus_constraint_rule**(*backer*
loc_tec
timeste

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

$\boldsymbol{storage}(loc :: tech, timestep) = \boldsymbol{storage}(loc :: tech, timestep_{previous}) \times (1 - storage\_loss(loc :: tech, timestep))^{timest}$

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

$\boldsymbol{resource_{con}}(loc :: tech, timestep) \times \eta_{resource}(loc :: tech, timestep) = \dfrac{\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, time}{\eta_{energy}(loc :: tech, timestep) \times \eta_{parasitic}(loc :: te}$

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

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

$\boldsymbol{storage}(loc :: tech, timestep) = \boldsymbol{storage}(loc :: tech, timestep_{previous}) \times (1 - storage\_loss(loc :: tech, timestep))^{resolut}$

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:

$$\boldsymbol{storage_{cap}}(loc :: tech) \begin{cases} = storage_{cap,equals}(loc :: tech), & \text{if } storage_{cap,equals}(loc :: tech) \\ \leq storage_{cap,max}(loc :: tech), & \text{if } storage_{cap,max}(loc :: tech) \\ \text{unconstrained}, & \text{otherwise} \end{cases} \quad \forall loc :: tech \in loc :: techs_{store}$$

and (if `equals` not enforced):

$$\boldsymbol{storage_{cap}}(loc :: tech) \geq storage_{cap,min}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{store}$$

calliope.backend.pyomo.constraints.capacity.**energy_capacity_storage_constraint_rule**(*backend_*
loc_tech)

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

$$\boldsymbol{energy_{cap}}(loc :: tech) \leq \boldsymbol{storage_{cap}}(loc :: tech) \times charge\_rate(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{store}$$

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:

$$\boldsymbol{resource_{cap}}(loc :: tech) \begin{cases} = resource_{cap,equals}(loc :: tech), & \text{if } resource_{cap,equals}(loc :: tech) \\ \leq resource_{cap,max}(loc :: tech), & \text{if } resource_{cap,max}(loc :: tech) \\ \text{unconstrained}, & \text{otherwise} \end{cases} \quad \forall loc :: tech \in loc :: techs_{f}$$

and (if `equals` not enforced):

$$\boldsymbol{resource_{cap}}(loc :: tech) \geq resource_{cap,min}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{finite\_resource\_supply\_plus}$$

calliope.backend.pyomo.constraints.capacity.**resource_capacity_equals_energy_capacity_const**

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

$$\boldsymbol{resource_{cap}}(loc :: tech) = \boldsymbol{energy_{cap}}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{finite\_resource\_supply\_plus} \text{ if } resource\_cap\_$$

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:

$$\boldsymbol{resource_{area}}(loc :: tech) \begin{cases} = resource_{area,equals}(loc :: tech), & \text{if } resource_{area,equals}(loc :: tech) \\ \leq resource_{area,max}(loc :: tech), & \text{if } resource_{area,max}(loc :: tech) \\ \text{unconstrained}, & \text{otherwise} \end{cases} \quad \forall loc :: tech \in loc :: tec$$

and (if `equals` not enforced):

$$\boldsymbol{resource_{area}}(loc :: tech) \geq resource_{area,min}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{area}$$

calliope.backend.pyomo.constraints.capacity.**resource_area_per_energy_capacity_constraint_ru**

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

$$\boldsymbol{resource_{area}}(loc :: tech) = \boldsymbol{energy_{cap}}(loc :: tech) \times area\_per\_energy\_cap(loc :: tech) \quad \forall loc :: tech \in locs :: techs_{are}$$

calliope.backend.pyomo.constraints.capacity.**resource_area_capacity_per_loc_constraint_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

$$\sum_{tech} \boldsymbol{resource_{area}}(loc :: tech) \leq available\_area \quad \forall loc \in locs \text{ if } available\_area(loc)$$

calliope.backend.pyomo.constraints.capacity.**energy_capacity_constraint_rule**(*backend_model*, *loc_tech*)

Set upper and lower bounds for energy_cap.

The first valid case is applied:

$$\frac{\boldsymbol{energy_{cap}}(loc :: tech)}{energy_{cap,scale}(loc :: tech)} \begin{cases} = energy_{cap,equals}(loc :: tech), & \text{if } energy_{cap,equals}(loc :: tech) \\ \leq energy_{cap,max}(loc :: tech), & \text{if } energy_{cap,max}(loc :: tech) \quad \forall loc :: tech \in loc :: techs \\ \text{unconstrained}, & \text{otherwise} \end{cases}$$

and (if `equals` not enforced):

$$\frac{\boldsymbol{energy_{cap}}(loc :: tech)}{energy_{cap,scale}(loc :: tech)} \geq energy_{cap,min}(loc :: tech) \quad \forall loc :: tech \in loc :: techs$$

calliope.backend.pyomo.constraints.capacity.**energy_capacity_systemwide_constraint_rule**(*backe*
tech)

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

The first valid case is applied:

$$\sum_{loc} \boldsymbol{energy_{cap}}(loc :: tech) \begin{cases} = energy_{cap,equals,systemwide}(loc :: tech), & \text{if } energy_{cap,equals,systemwide}(loc :: tech) \\ \leq energy_{cap,max,systemwide}(loc :: tech), & \text{if } energy_{cap,max,systemwide}(loc :: tech) \quad \forall tech \in \\ \text{unconstrained}, & \text{otherwise} \end{cases}$$

Dispatch

calliope.backend.pyomo.constraints.dispatch.**carrier_production_max_constraint_rule**(*backend_m*
loc_tech_c
timestep)

Set maximum carrier production. All technologies.

$$\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \leq energy_{cap}(loc :: tech) \times timestep\_resolution(timestep) \times parasitic\_eff$$

calliope.backend.pyomo.constraints.dispatch.**carrier_production_min_constraint_rule**(*backend_m*
loc_tech_c
timestep)

Set minimum carrier production. All technologies except `conversion_plus`.

$$\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \geq energy_{cap}(loc :: tech) \times timestep\_resolution(timestep) \times energy_{cap,min}$$

calliope.backend.pyomo.constraints.dispatch.**carrier_consumption_max_constraint_rule**(*backend_loc_tech_timestep*)

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

$$\boldsymbol{carrier_{con}}(loc :: tech :: carrier, timestep) \geq -1 \times energy_{cap}(loc :: tech) \times timestep\_resolution(timestep)$$

calliope.backend.pyomo.constraints.dispatch.**resource_max_constraint_rule**(*backend_model*, *loc_tech*, *timestep*)

Set maximum resource consumed by supply_plus techs.

$$\boldsymbol{resource_{con}}(loc :: tech, timestep) \leq timestep\_resolution(timestep) \times resource_{cap}(loc :: tech)$$

calliope.backend.pyomo.constraints.dispatch.**storage_max_constraint_rule**(*backend_model*, *loc_tech*, *timestep*)

Set maximum stored energy. Supply_plus & storage techs only.

$$\boldsymbol{storage}(loc :: tech, timestep) \leq storage_{cap}(loc :: tech)$$

calliope.backend.pyomo.constraints.dispatch.**ramping_up_constraint_rule**(*backend_model*, *loc_tech_carrier*, *timestep*)

Ramping up constraint.

$$diff(loc :: tech :: carrier, timestep) \leq max\_ramping\_rate(loc :: tech :: carrier, timestep)$$

calliope.backend.pyomo.constraints.dispatch.**ramping_down_constraint_rule**(*backend_model*, *loc_tech_carrier*, *timestep*)

Ramping down constraint.

$$-1 \times max\_ramping\_rate(loc :: tech :: carrier, timestep) \leq diff(loc :: tech :: carrier, timestep)$$

calliope.backend.pyomo.constraints.dispatch.**ramping_constraint**(*backend_model*, *loc_tech_carrier*, *timestep*, *direction=0*)

Ramping rate constraints.

Direction: 0 is up, 1 is down.

$$\boldsymbol{diff}(loc :: tech :: carrier, timestep) = (carrier_{prod}(loc :: tech :: carrier, timestep) + carrier_{con}(loc :: tech :: carrier, ti$$

Costs

calliope.backend.pyomo.constraints.costs.**cost_constraint_rule**(*backend_model*, *cost*, *loc_tech*)

Combine investment and time varying costs into one cost per technology

$$\boldsymbol{cost}(cost, loc :: tech) = \boldsymbol{cost_{investment}}(cost, loc :: tech) + \sum_{timestep \in timesteps} \boldsymbol{cost_{var}}(cost, loc :: tech, timestep)$$

calliope.backend.pyomo.constraints.costs.**cost_investment_constraint_rule**(*backend_model*, *cost*, *loc_tech*)

Calculate costs from capacity decision variables.

Transmission technologies "exist" at two locations, so their cost is divided by 2.

$$cost_{con}(cost, loc :: tech) = depreciation\_rate * ts\_weight * (cost_{energy\_cap}(cost, loc :: tech) \times \boldsymbol{energy_{cap}}(loc :: tech) + c$$

calliope.backend.pyomo.constraints.costs.**cost_var_constraint_rule**(*backend_model*, *cost*, *loc_tech*, *timestep*)

Calculate costs from time-varying decision variables

$$\boldsymbol{cost_{var}}(cost, loc :: tech, timestep) = cost_{prod}(cost, loc :: tech, timestep) + cost_{con}(co$$

$$cost_{prod}(cost, loc :: tech, timestep) = cost_{om\_prod}(cost, loc :: tech, timestep) \times weight(timestep) \times \boldsymbol{carrier_{prod}}(loc :: te$$

$$prod\_con\_eff = \begin{cases} = \boldsymbol{resource_{con}}(loc :: tech, timestep), & \text{if } loc :: tech \in \\ = \frac{\boldsymbol{carrier_{prod}}(loc::tech::carrier,timestep)}{energy_eff(loc::tech,timestep)}, & \text{if } loc :: tech \in \end{cases}$$

$$cost_{con}(cost, loc :: tech, timestep) = cost_{om\_con}(cost, loc :: tech, timestep) \times weight(tim$$

Export

calliope.backend.pyomo.constraints.export.**update_system_balance_constraint**(*backend_model*, *loc_carrier*, *timestep*)

Update system balance constraint (from energy_balance.py) to include export

Math given in *system_balance_constraint_rule()*

calliope.backend.pyomo.constraints.export.**export_balance_constraint_rule**(*backend_model*, *loc_tech_carrier*, *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

$$\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \geq \boldsymbol{carrier_{export}}(loc :: tech :: carrier, timestep) \quad \forall loc :: tech :: carrier \in$$

calliope.backend.pyomo.constraints.export.**update_costs_var_constraint**(*backend_model*, *cost*, *loc_tech*, *timestep*)

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

$$\boldsymbol{cost_{var}}(cost, loc :: tech, timestep) += cost_{export}(cost, loc :: tech, timestep) \times \boldsymbol{carrier_{export}}(loc :: tech :: carrier, time$$

calliope.backend.pyomo.constraints.export.**export_max_constraint_rule**(*backend_model*, *loc_tech_carrier*, *timestep*)

Set maximum export. All exporting technologies.

$$\boldsymbol{carrier_{export}}(loc :: tech :: carrier, timestep) \leq export_{cap}(loc :: tech) \quad \forall loc :: tech :: carrier \in locs :: tech :: carriers_{ex}$$

If the technology is defined by integer units, not a continuous capacity, this constraint becomes:

$$\boldsymbol{carrier_{export}}(loc :: tech :: carrier, timestep) \leq export_{cap}(loc :: tech) \times \boldsymbol{operating_{units}}(loc :: tech, timestep)$$

MILP

calliope.backend.pyomo.constraints.milp.**unit_commitment_constraint_rule**(*backend_model*, *loc_tech*, *timestep*)

Constraining the number of integer units $operating_units(loc_tech, timestep)$ of a technology which can operate in a given timestep, based on maximum purchased units $units(loc_tech)$

$$\boldsymbol{operating\_units}(loc :: tech, timestep) \leq \boldsymbol{units}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{milp}, \forall timestep \in timesteps$$

calliope.backend.pyomo.constraints.milp.**unit_capacity_constraint_rule**(*backend_model*, *loc_tech*)

Add upper and lower bounds for purchased units of a technology

$$\boldsymbol{units}(loc :: tech) \begin{cases} = units_{equals}(loc :: tech), & \text{if } units_{equals}(loc :: tech) \\ \leq units_{max}(loc :: tech), & \text{if } units_{max}(loc :: tech) \\ \text{unconstrained}, & \text{otherwise} \end{cases} \quad \forall loc :: tech \in loc :: techs_{milp}$$

and (if `equals` not enforced):

$$\boldsymbol{units}(loc :: tech) \geq units_{min}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{milp}$$

calliope.backend.pyomo.constraints.milp.**carrier_production_max_milp_constraint_rule**(*backend_*, *loc_tech_*, *timestep*)

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

$$\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \leq energy_{cap,perunit}(loc :: tech) \times timestep\_resolution(timestep) \times \boldsymbol{opera}$$

$\eta_{parasitic}$ is only activated for *supply_plus* technologies

calliope.backend.pyomo.constraints.milp.**carrier_production_max_conversion_plus_milp_constra**

Set maximum carrier production of conversion_plus MILP techs

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{out}} \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \leq energy_{cap,perunit}(loc :: tech) \times timeste$$

calliope.backend.pyomo.constraints.milp.**carrier_production_min_milp_constraint_rule**(*backend_*, *loc_tech_*, *timestep*)

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

$$\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \geq energy_{cap,perunit}(loc :: tech) \times timestep\_resolution(timestep) \times \boldsymbol{opera}$$

calliope.backend.pyomo.constraints.milp.**carrier_production_min_conversion_plus_milp_constra**

Set minimum carrier production of conversion_plus MILP techs

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{out}} \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \geq energy_{cap,perunit}(loc :: tech) \times timeste$$

calliope.backend.pyomo.constraints.milp.**carrier_consumption_max_milp_constraint_rule**(*backend*
loc_tech
timestep

Set maximum carrier consumption of demand, storage, and transmission MILP techs

$$\boldsymbol{carrier_{con}}(loc :: tech :: carrier, timestep) \geq -1 * energy_{cap,perunit}(loc :: tech) \times timestep\_resolution(timestep) \times \boldsymbol{op}$$

calliope.backend.pyomo.constraints.milp.**energy_capacity_units_constraint_rule**(*backend_model*,
loc_tech)

Set energy capacity decision variable as a function of purchased units

$$\boldsymbol{energy_{cap}}(loc :: tech) = \boldsymbol{units}(loc :: tech) \times energy_{cap,perunit}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{milp}$$

calliope.backend.pyomo.constraints.milp.**storage_capacity_units_constraint_rule**(*backend_model*,
loc_tech)

Set storage capacity decision variable as a function of purchased units

$$\boldsymbol{storage_{cap}}(loc :: tech) = \boldsymbol{units}(loc :: tech) \times storage_{cap,perunit}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{milp,store}$$

calliope.backend.pyomo.constraints.milp.**energy_capacity_max_purchase_constraint_rule**(*backend*
loc_tech

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

The first valid case is applied:

$$\frac{\boldsymbol{energy_{cap}}(loc :: tech)}{energy_{cap,scale}(loc :: tech)} \begin{cases} = energy_{cap,equals}(loc :: tech) \times \boldsymbol{purchased}(loc :: tech), & \text{if } energy_{cap,equals}(loc :: tech) \\ \leq energy_{cap,max}(loc :: tech) \times \boldsymbol{purchased}(loc :: tech), & \text{if } energy_{cap,max}(loc :: tech) \quad \forall \\ \text{unconstrained}, & \text{otherwise} \end{cases}$$

calliope.backend.pyomo.constraints.milp.**energy_capacity_min_purchase_constraint_rule**(*backend*
loc_tech

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

and (if `equals` not enforced):

$$\frac{\boldsymbol{energy_{cap}}(loc :: tech)}{energy_{cap,scale}(loc :: tech)} \geq energy_{cap,min}(loc :: tech) \times \boldsymbol{purchased}(loc :: tech) \quad \forall loc :: tech \in loc :: techs$$

calliope.backend.pyomo.constraints.milp.**storage_capacity_max_purchase_constraint_rule**(*backer*
loc_te

Set maximum storage capacity.

The first valid case is applied:

$$\boldsymbol{storage_{cap}}(loc :: tech) \begin{cases} = storage_{cap,equals}(loc :: tech) \times \boldsymbol{purchased}, & \text{if } storage_{cap,equals} \\ \leq storage_{cap,max}(loc :: tech) \times \boldsymbol{purchased}, & \text{if } storage_{cap,max}(loc :: tech) \; \forall loc :: tech \in lo \\ \text{unconstrained}, & \text{otherwise} \end{cases}$$

calliope.backend.pyomo.constraints.milp.**storage_capacity_min_purchase_constraint_rule**(*backer*
loc_te

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

if `equals` not enforced for storage_cap:

$$\boldsymbol{storage_{cap}}(loc :: tech) \geq storage_{cap,min}(loc :: tech) \times \boldsymbol{purchased}(loc :: tech) \quad \forall loc :: tech \in loc :: techs_{purchase,store}$$

calliope.backend.pyomo.constraints.milp.**update_costs_investment_units_constraint**(*backend_mod*
cost,
loc_tech)

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

$$\boldsymbol{cost_{investment}}(cost, loc :: tech) += \boldsymbol{units}(loc :: tech) \times cost_{purchase}(cost, loc :: tech) * timestep_{weight} * depreciation$$

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

Add binary investment costs (cost * binary_purchased_unit)

$$\boldsymbol{cost_{investment}}(cost, loc :: tech) += \boldsymbol{purchased}(loc :: tech) \times cost_{purchase}(cost, loc :: tech) * timestep_{weight} * depreci$$

Conversion

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

Balance energy carrier consumption and production

$$-1 * \boldsymbol{carrier_{con}}(loc :: tech :: carrier, timestep) \times \eta_{energy}(loc :: tech, timestep) = \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, ti$$

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

Add time-varying conversion technology costs

$$\boldsymbol{cost_{var}}(loc :: tech, cost, timestep) = \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \times timestep_{weight}(timestep) \times cost_{om}$$

Conversion_plus

calliope.backend.pyomo.constraints.conversion_plus.**balance_conversion_plus_primary_constra**

Balance energy carrier consumption and production for carrier_in and carrier_out

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{out}} \frac{\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep)}{carrier\_ratio(loc :: tech :: carrier, `out')} = -1 * \sum_{loc::tech::carrier \in loc::tech::carriers_{in}} (c$$

calliope.backend.pyomo.constraints.conversion_plus.**carrier_production_max_conversion_plus_c**

Set maximum conversion_plus carrier production.

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{out}} \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \leq \boldsymbol{energy_{cap}}(loc :: tech) \times timestep\_res$$

calliope.backend.pyomo.constraints.conversion_plus.**carrier_production_min_conversion_plus_c**

Set minimum conversion_plus carrier production.

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{out}} \boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep) \leq \boldsymbol{energy_{cap}}(loc :: tech) \times timestep\_res$$

calliope.backend.pyomo.constraints.conversion_plus.**cost_var_conversion_plus_constraint_rule**

Add time-varying conversion_plus technology costs

$$\boldsymbol{cost_{var}}(loc :: tech, cost, timestep) = \boldsymbol{carrier_{prod}}(loc :: tech :: carrier_{primary}, timestep) \times timestep_{weight}(timestep) \times$$

calliope.backend.pyomo.constraints.conversion_plus.**balance_conversion_plus_tiers_constraint**

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 ['out_2', 'out_3']:

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{out}} (\frac{\boldsymbol{carrier_{prod}}(loc :: tech :: carrier, timestep)}{carrier\_ratio(loc :: tech :: carrier, `out')} = \sum_{loc::tech::carrier \in loc::tech::carriers_{tier}} (\frac{\boldsymbol{car}}{c}$$

If *tier* in ['in_2', 'in_3']:

$$\sum_{loc::tech::carrier \in loc::tech::carriers_{in}} \frac{\boldsymbol{carrier_{con}}(loc :: tech :: carrier, timestep)}{carrier\_ratio(loc :: tech :: carrier, `in')} = \sum_{loc::tech::carrier \in loc::tech::carriers_{tier}} \frac{\boldsymbol{carri}}{carr}$$

Network

calliope.backend.pyomo.constraints.network.**symmetric_transmission_constraint_rule**(*backend_mo*
loc_tech)

Constrain e_cap symmetrically for transmission nodes. Transmission techs only.

$$energy_{cap}(loc1 :: tech : loc2) = energy_{cap}(loc2 :: tech : loc1)$$

Policy

calliope.backend.pyomo.constraints.policy.**group_share_energy_cap_constraint_rule**(*backend_mod*
tech-
list,
what)

Enforce shares in energy_cap for groups of technologies. Applied to `supply` and `supply_plus` technologies only.

$$\sum_{loc::tech \in given\_group} energy_{cap}(loc :: tech) = fraction \times \sum_{loc::tech \in loc\_techs\_supply loc\_techs\_supply\_plus} energy_{cap}(loc :: tech)$$

calliope.backend.pyomo.constraints.policy.**group_share_carrier_prod_constraint_rule**(*backend_m*
tech-
list_carrie
what)

Enforce shares in carrier_prod for groups of technologies. Applied to `loc_tech_carriers_supply_all`, which includes supply, supply_plus, conversion, and conversion_plus.

$$\sum_{loc::tech::carrier \in given\_group, timestep \in timesteps} carrier_{prod}(loc :: tech :: carrier, timestep) = fraction \times \sum_{loc::tech:carrier \in loc\_te}$$

calliope.backend.pyomo.constraints.policy.**reserve_margin_constraint_rule**(*backend_model*, *car-*
rier)

Enforces a system reserve margin per carrier.

$$\sum_{loc::tech::carrier \in loc\_tech\_carriers\_supply\_all} energy_{cap}(loc :: tech :: carrier, timestep_{max\_demand}) \geq \sum_{loc::tech::carrier \in loc\_tech\_ca}$$

1.9 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 `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.

1.9.1 Installing a development version

As when installing a stable version, using `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):

```
$ 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:

```
$ 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`:

```
$ conda env create -f ./calliope/requirements/base.yml -n calliope_dev
$ source activate calliope_dev
```

On Windows:

```
$ conda env create -f ./calliope/requirements/base.yml -n calliope_dev
$ activate calliope_dev
```

Then install Calliope itself with pip:

```
$ pip install -e ./calliope
```

1.9.2 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:

```
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: {...}
```

1.9.3 Profiling

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

```
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`)

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

- Generate a call graph from the call tree via graphviz

  ```
  # brew install gprof2dot
  gprof2dot -f callgrind calliope.calltree | dot -Tsvg -o callgraph.svg
  ```

1.9.4 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" %}`
    - MD5 of latest version from PyPI: `{% set md5 = "XXXX" %}`
    - Reset `build: number: 0` if it is not already at zero
    - If necessary, carry over any changed requirements from `requirements.yml` or `setup.py`
  - 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

dropna [bool, optional] If True (default), NaN values are dropped when saving, resulting in significantly smaller CSV files.

2.1.2 Time series

calliope.core.time.clustering.**get_clusters**(*data*, *func*, *timesteps_per_day*, *tech=None*, *timesteps=None*, *k=None*, *variables=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.

kwargs [dict] Additional keyword arguments available depend on the *func*. For available KMeans kwargs see: http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html For available hierarchical kwargs see: http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

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'*, *n=1*, *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 diffence 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**(*source_dict=None*)

A subclass of `dict` with key access by attributes:

```
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(*deep=False*)

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:

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

classmethod from_yaml(*f*, *resolve_imports=True*)

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

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

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

classmethod from_yaml_string(*string*)

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.:

```
d = AttrDict()
d.set_key('foo.bar', 1)
d.foo.bar == 1  # True
```

get_key(*key*, *default=MISSING*)
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*, *convert_objects=True*, *format_lists=True*, ***kwargs*)
Saves the AttrDict to the given path as a YAML file.

If `path` is None, returns the YAML string instead.

Any additional keyword arguments are passed to the YAML writer, so can use e.g. `indent=4` to override the default of 2.

`convert_objects` (defaults to True) controls whether Numpy objects should be converted to regular Python objects, so that they are properly displayed in the resulting YAML output.

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': ['b1', '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 0.6.1 (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

3.1.2 0.6.0 (2018-04-20)

Version 0.6.0 is an almost complete rewrite of most of Calliope's internals. See *New in v0.6.0* 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. See *0.6.0 model configuration changes* for a full list of changes.

new backwards-incompatible Complete rewrite of Pyomo backend, including new various new and improved functionality to interact with a built model (see *New in v0.6.0*).

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`. See *0.6.0 model configuration changes* for a full list.

changed backwards-incompatible `system_margin` constraint replaced with `reserve_margin` constraint.

changed backwards-incompatible Removed the ability to load additional custom constraints or objectives.

3.1.3 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

3.1.4 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.

3.1.5 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)

3.1.6 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.

3.1.7 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

3.1.8 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 * 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

3.1.9 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

3.1.10 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

3.1.11 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

3.1.12 0.3.6 (2015-09-23)

- fixed Version 0.3.5 changes were not reflected in tutorial

3.1.13 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`:

- `constraints.group_fraction.group_fraction` becomes `constraints.optional.group_fraction`
- `constraints.ramping.ramping_rate` becomes `constraints.optional.ramping_rate`

Other changes

- new analysis.map_results function to extract solution details from multiple parallel runs
- new Various other additions to analysis functionality, particularly in the analysis_utils module
- new 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 `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

3.1.14 0.3.4 (2015-04-27)

- fixed Bug in construction and fixed O&M cost calculations in operational mode

3.1.15 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 `time.masks`, with a base resolution via `time.resolution` (or instead, as before, load a resolution series from file via `time.file`). Renamed the `time_functions` submodule to `time_masks`.

Other changes

- new Models and runs can have a `name`
- changed More verbose `calliope run`
- changed Analysis tools restructured
- changed Renamed `debug.keepfiles` setting to `debug.keep_temp_files` and better documented debug configuration

3.1.16 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

3.1.17 0.3.1 (2015-01-06)

- Fixes to time_functions
- Other minor improvements and fixes

3.1.18 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

3.1.19 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

3.1.20 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

Release history

CHAPTER 4

License

Bibliography

[Fripp2012] Fripp, M., 2012. Switch: A Planning Tool for Power Systems with Large Shares of Intermittent Renewable Energy. Environ. Sci. Technol., 46(11), p.6371–6378. DOI: 10.1021/es204645c

[Heussen2010] Heussen, K. et al., 2010. Energy storage in power system operation: The power nodes modeling framework. In Innovative Smart Grid Technologies Conference Europe (ISGT Europe), 2010 IEEE PES. pp. 1–8. DOI: 10.1109/ISGTEUROPE.2010.5638865

[Howells2011] Howells, M. et al., 2011. OSeMOSYS: The Open Source Energy Modeling System: An introduction to its ethos, structure and development. Energy Policy, 39(10), p.5850–5870. DOI: 10.1016/j.enpol.2011.06.033

[Hunter2013] Hunter, K., Sreepathi, S. & DeCarolis, J.F., 2013. Modeling for insight using Tools for Energy Model Optimization and Analysis (Temoa). Energy Economics, 40, p.339–349. DOI: 10.1016/j.eneco.2013.07.014

Python Module Index

C

Index

D

E

F

G

I

K

M

N

O

P

R

S

T

U