Calliope Documentation

Release 0.5.2

Stefan Pfenninger

Jun 16, 2017

Contents

v0.5.2 (*Release history*)

Calliope is a framework to develop energy system models using a modern and open source Python-based toolchain.

This is the documentation for version 0.5.2. See the main project website for contact details and other useful information.

Calliope is a framework to develop energy system models, with a focus 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 optimization 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 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.

Main features:

- Generic technology definition allows modeling 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
- Model specification in an easy-to-read and machine-processable YAML format
- Able to run on computing clusters
- Easily extensible in a modular way: custom constraint generator functions and custom time mask functions
- 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

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 modeling 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 modeling systems are available under open source or free software licenses:

- SWITCH: A power system model focused on renewables integration, using multi-stage stochastic linear optimization, 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 optimization 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.

1.1.1 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

1.1.2 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.

1.1.3 License

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

```
Copyright 2013-2017 Calliope contributors listed in AUTHORS

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

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

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

1.1.4 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 3.5 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, CPLEX, and Gurobi. Any other solver that is compatible with Pyomo, which Calliope uses to construct its models, should 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 Anaconda Python distribution and its package manager, `conda`.

With Anaconda installed, you can create a new Python 3.5 environment called "calliope" with all the necessary modules, including the free and open source GLPK solver, with the following command:

```
$ conda create -c conda-forge -n calliope python=3.5 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. GLPK or Gurobi are recommended and have been confirmed to work with Calliope. Refer to the documentation of your solver on how to install it. Some details on GLPK and Gurobi are given below. Another commercial alternative is CPLEX.

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.

Gurobi

Gurobi is commercial but significantly faster than GLPK, 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).

1.2.4 Python module requirements

The following Python modules and their dependencies are required:

- Pyomo
- Pandas
- Xarray
- NetCDF4
- Numexpr
- PyYAML
- Click

Matplotlib is optional but necessary to graphically display results.

These modules are optional but necessary to display transmission flows on a map:

- NetworkX
- Basemap

These modules are optional and used for the example notebook in the tutorial:

- Seaborn
- Jupyter

1.3 Components to build a model

This section proves an overview of how a model is built using Calliope.

Calliope allows a modeler to define technologies with arbitrary characteristics by "inheriting" basic traits from a number of included base technologies, *which are described below*. Technologies can take a **resource** from outside of the modeled system and turn it into a specific energy **carrier** in the system. These technologies, together with the **locations** specified in the model, result in a set of **nodes**: the energy balance equations indexed over the set of technologies and locations.

1.3.1 Terminology

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

- **Technology**: a technology that produces, consumes, converts or transports energy
- **Location**: a site which can contain multiple technologies and which may contain other locations for energy balancing purposes
- **Node**: a combination of technology and location resulting in specific energy balance equations (*see below*)
- **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 optimization, the following terms are also used:

- Parameter: a fixed coefficient that enters into model equations

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

1.3.2 Index sets

Most parameters, variables, and constraints are formulated with respect to at least some of the indices below:

- `c`: carriers
- `y`: technologies
- `x`: locations
- `t`: time steps
- `k`: cost classes

In some cases, these index sets may have only a single member. For example, if only the power system is modeled, the set `c` (carriers) will have a single member, `power`.

1.3.3 Technology types

Each technology (that is, each member of the set `y`) is of a specific *technology type*, which determines how the framework models the technology and what properties it can have. The technology type is specified by inheritance from one of seven abstract base technologies (see *Technologies* in the model configuration section for more details on this inheritance model):

- Supply: Supplies energy from a resource to a carrier (a source) (base technology: `supply`)
- Supply_plus: A more feature rich version of `supply`. It can have storage of resource before conversion to carrier, can define an additional secondary resource, and can have several more intermediate loss factors (base technology: `supply_plus`)
- Demand: Acts like supply but with a resource that is negative (a sink). Draws energy from a carrier to satisfy a resource demand (base technology: `demand`)
- Conversion: Converts energy from one carrier to another, can have neither resource nor storage associated with it (base technology: `conversion`)
- Conversion_plus: A more feature rich version of `conversion`. There can be several carriers in, converted to several carriers out (base technology: `conversion_plus`)
- Storage: Can store energy of a specific carrier, cannot have any resource (base technology: `storage`)
- Transmission: Transports energy of a specific carrier from one location to another, can have neither resource nor storage (base technology: `transmission`)

The internal definition of these abstract base technologies is given in the *configuration reference*.

1.3.4 Cost classes

Costs are modeled in Calliope via *cost classes*. By default, only one classes is defined: `monetary`.

Technologies can define costs for components (installed capacity), for operation & maintenance, and for export for any cost class. Costs can be given as negative values, which defines a revenue rather than a cost.

The primary cost class, `monetary`, is used to calculate levelized costs and by default enters into the objective function. Therefore each technology should define at least one type of `monetary` cost, as it would be considered free otherwise. By default, any cost not specified is assumed to be zero.

Only the `monetary` cost class is entered into the default objective function, but other cost classes can be defined for accounting purposes, e.g. `emissions` to account for greenhouse gas emissions. Additional cost classes can be created simply by adding them to the definition of costs for a technology (see the *model configuration section* for more detail on this).

Revenue

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.

1.3.5 Putting technologies and locations together: Nodes

In the model definition, locations can be defined, and for each location (or for groups of locations), technologies can be permitted. The details of this are laid out in the *model configuration section*.

A *node* is the combination of a specific location and technology, and is how Calliope internally builds the model. For a given location, `x`, and technology, `y`, a set of equations defined over `(x, y)` models that specific node.

The most important node variables are laid out below, but more detail is also available in the section *Model formulation*.

1.3.6 Node energy balance

The basic formulation of each node uses a set of energy balance equations. Depending on the technology type, different energy balance variables are used:

- **`s(y, x, t)`: storage level at time `t`** This is used for `storage` and `supply_plus` technologies.
- **`r(y, x, t)`: resource to technology (+ production) at time `t`. If storage is defined for `supply_plus`, this is resource to** This is used for `supply_plus` technologies.
- **`r2(y, x, t)`: secondary resource to technology at time `t`** This is used for `supply_plus` technologies.
- **`c_prod(c, y, x, t)`: production of a given energy carrier by a technology (+ supply) at time `t`.** This is used for all technologies, except `demand`.
- **`c_con(c, y, x, t)`: consumption of a given energy carrier by a technology at time `t`** This is used for all technologies, except `supply` and `supply_plus`.

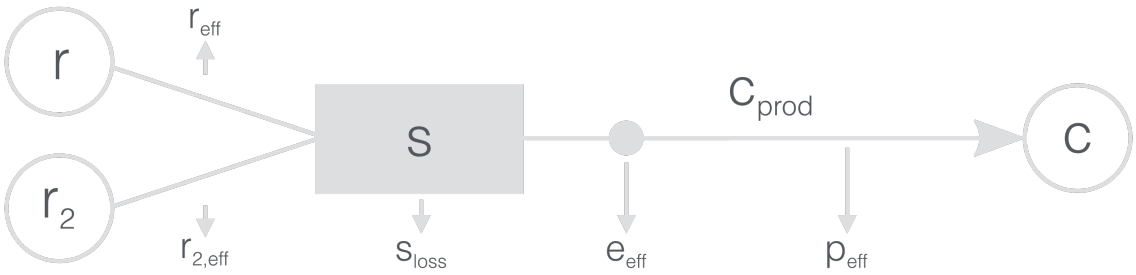
The resulting losses associated with energy balancing also depend on the technology type. Each technology node is mapped here, with details on interactions given in *Model configuration*.

The secondary resource can deliver energy to storage via `r_2` alongside the primary energy source (via `r`), but only if the necessary setting (`constraints.allow_r2:`) is enabled for a technology. Optionally, this can be allowed only during the `startup_time:` (defined in the model-wide settings), e.g. to allow storage to be filled up initially.

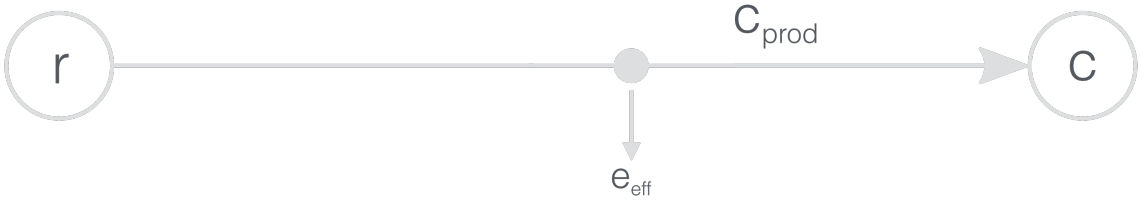
Each node can also have the following capacity variables:

- **`s_cap(y, x)`: installed storage capacity** This is used for `storage` and `supply_plus` technologies.
- **`r_cap(y, x)`: installed resource to storage conversion capacity** This is used for `supply_plus` technologies.
- **`r_area(y, x)`: installed resource collector area** This is used for `supply`, `supply_plus`, and `demand` technologies.

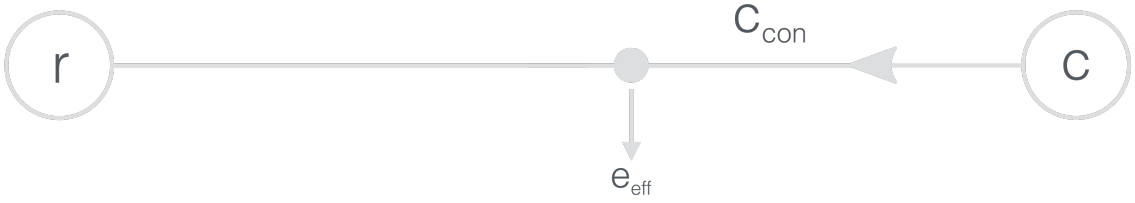
Supply+



Supply



Demand



- **`e_cap(y, x)`: installed storage to carrier conversion capacity** This is used for all technologies,.
- **`r2_cap(y, x)`: installed secondary resource to storage conversion capacity** This is used for `supply_plus` technologies.

Note: For nodes that have an internal (parasitic) energy consumption, `e_cap_net` is also included in the solution. This specifies the net conversion capacity, while `e_cap(y, x)` is gross capacity.

When defining a technology, it must be given at least some constraints, that is, options that describe the functioning of the technology. If not specified, all of these are inherited from the default technology definition (with default values being `0` for capacities and `1` for efficiencies). Some examples of such options are:

- `resource(y, x, t)`: available resource (+ source, - sink)
- `s_cap.max(y)`: maximum storage capacity
- `s_loss(y, t)`: storage loss rate
- `r_area.max(y)`: maximum resource collector area
- `r_eff(y)`: resource efficiency
- `r_cap.max(y)`: maximum resource to storage conversion capacity
- `e_eff(y, t)`: resource/storage/carrier_in to carrier_out conversion efficiency
- `e_cap.max(y)`: maximum installed carrier conversion capacity, applied to carrier_out

---

Note: Generally, these constraints are defined on a per-technology basis. However, some (but not all) of them may be overridden on a per-location basis. This allows, for example, setting different constraints on the allowed maximum capacity for a specific technology at each location separately. See *Model configuration* for details on this.

Finally, each node tracks its costs (+ costs, - revenue), formulated in two constraints (more details in the *Model formulation* section):

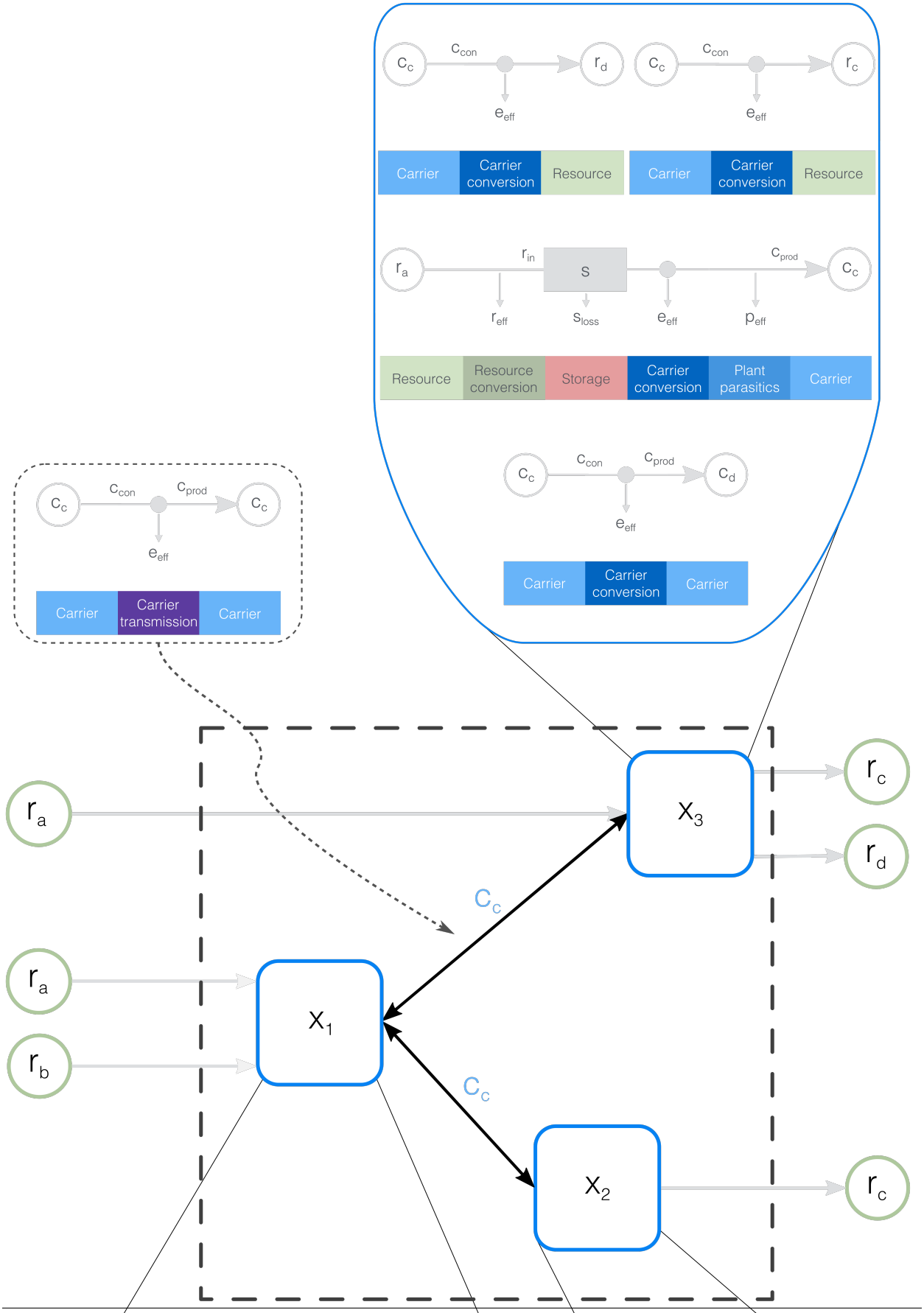
- `cost_fixed`: construction and fixed operational and maintenance (O&M) costs (i.e., costs per unit of installed capacity)
- `cost_var`: variable O&M and export costs (i.e., costs per produced unit of output)

Note: Efficiencies, available resources, and costs can be defined to vary in time. Equally (and more likely) they can be given as single values. For more detail on time-varying versus constant values, see *the corresponding section* in the model formulation chapter.

1.3.7 Linking locations

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.

Transmission links are considered by the system as nodes at each end of the link, with the same technology at each end. In this regard, the same nodal energy balance equations apply. Additionally, the user can utilise per-distance constraints and costs. For more information on available constraints/costs, see the *Model configuration* section.


c_c
c_{con}
r_d
e_{eff}
Carrier
Carrier conversion
Resource
c_c
c_{con}
r_c
e_{eff}
Carrier
Carrier conversion
Resource
r_a
r_{in}
s
c_{prod}
c_c
r_{eff}
s_{loss}
e_{eff}
p_{eff}
Resource
Resource conversion
Storage
Carrier conversion
Plant parasitics
Carrier
c_c
c_{con}
c_{prod}
c_d
e_{eff}
Carrier
Carrier conversion
Carrier
c_c
c_{con}
c_{prod}
c_c
e_{eff}
Carrier
Carrier transmission
Carrier
r_a
X_3
r_c
r_d
C_c
r_a
X_1
r_b
C_c
X_2
r_c

The next section is a brief tutorial. Following this, *Model formulation* details the constraints that actually implement all these formulations mathematically. The section following it, *Model configuration*, details how a model is configured, and how the various components outlined here are defined in a working model.

1.4 Tutorials

Before going through these tutorials, it is recommended to have a brief look at the *components section* to become familiar with the terminology and modeling approach used.

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.

The first part of the 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 part of the 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

1.4.1 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:

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

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

```
    ccgt:
        name: 'Combined cycle gas turbine'
        color: '#FDC97D'
        stack_weight: 200
        parent: supply
        carrier_out: power
        constraints:
            r: inf
            e_eff: 0.5
            e_cap.max: 40000  # kW
        costs:
```

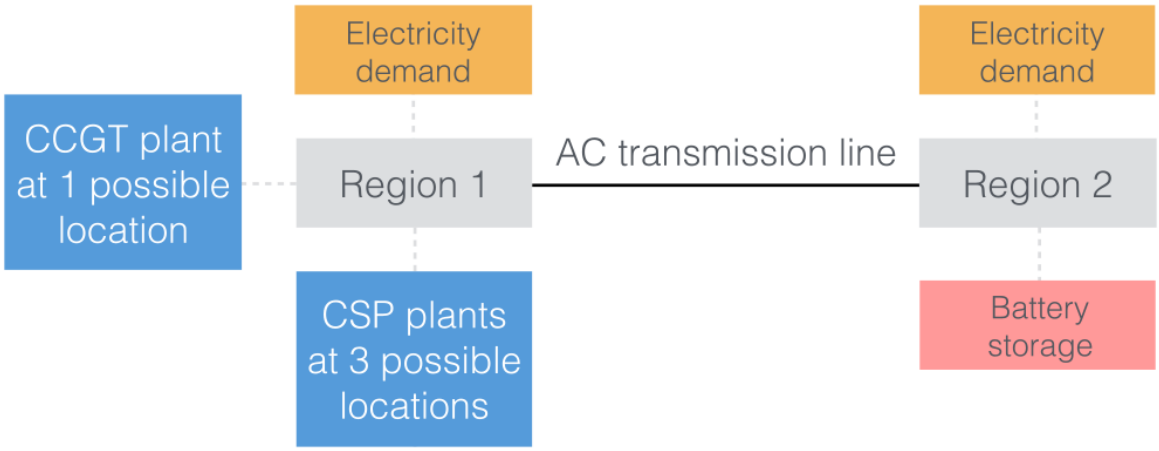


Fig. 1.3: Overview of the built-in national-scale example model

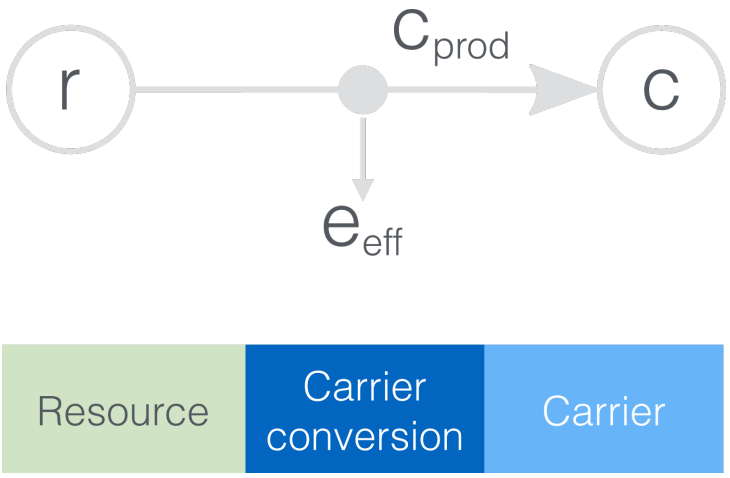


Fig. 1.4: The layout of a supply node, in this case `ccgt`, which has an infinite resource, a carrier conversion efficiency (e_{eff}), and a constraint on its maximum built e_{cap} (which puts an upper limit on e_{prod}).

```
            monetary:
                e_cap: 750  # USD per kW
                om_fuel: 0.02  # USD per kWh
```

There are a few things to note. First, `ccgt` 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, `supply`, and its carrier_out, `power`, thus setting 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 (`p_eff`)

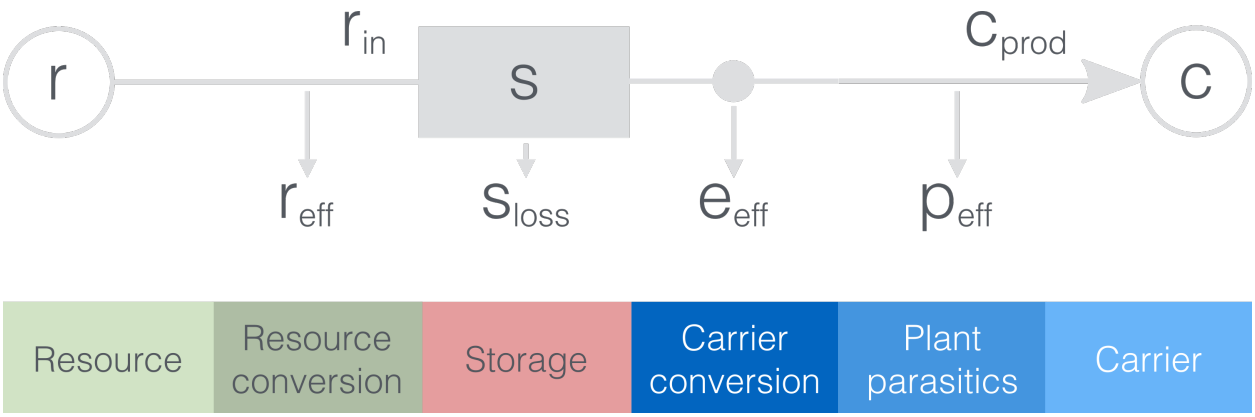


Fig. 1.5: The layout of a more complex node, in this case `csp`, which makes use of most node-level functionality available, with the exception of a secondary resource.

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

```
    csp:
        name: 'Concentrating solar power'
        color: '#99CB48'
        stack_weight: 100
        parent: supply_plus
        carrier_out: power
        constraints:
            use_s_time: true
            s_time.max: 24
            s_loss: 0.002
            r: file  # Will look for `csp_r.csv` in data directory
            e_eff: 0.4
            p_eff: 0.9
```

```
            r_area.max: inf
            e_cap.max: 10000
        costs:
            monetary:
                s_cap: 50
                r_area: 200
                r_cap: 200
                e_cap: 1000
                om_var: 0.002
        depreciation:
            monetary:
                interest: 0.12
```

Again, `csp` has the definitions for name, color, stack_weight, parent, and carrier_out. Its constraints are more numerous: it defines a maximum storage time (`s_time.max`), an hourly storage loss rate (`s_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:

```
    battery:
        name: 'Battery storage'
        color: '#DC5CE5'
        parent: storage
        carrier: power
        constraints:
            e_cap.max: 1000  # kW
            s_cap.max: inf
            c_rate: 4
            e_eff: 0.95  # 0.95 * 0.95 = 0.9025 round trip efficiency
            s_loss: 0  # No loss over time assumed
        costs:
            monetary:
                s_cap: 200  # USD per kWh storage capacity
```

The contraints give a maximum installed generation capacity for battery storage together with a charge rate (C-rate) of 4, which in turn limits the storage capacity. In the case of a storage technology, `e_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 and unmet power demand:

```
    demand_power:
        name: 'Power demand'
```

```
        parent: demand
        carrier: power
    unmet_demand_power:
        name: 'Unmet power demand'
        parent: unmet_demand
        carrier: power
```

Power demand is a technology like any other. We will associate an actual demand time series with the demand technology later. The parent of `unmet_demand_power`, `unmet_demand`, is a special kind of supply technology with an unlimited resource but very high cost. It allows a model to remain mathematically feasible even if insufficient supply is available to meet demand, and model results can easily be examined to verify whether there was any unmet demand. There is no requirement to include such a technology in a model, but it is useful to do so, since in its absence, an infeasible model would cause the solver to end with an error, returning no results for Calliope to analyze.

What remains to set up is a simple transmission technology:

```
    ac_transmission:
        name: 'AC power transmission'
        parent: transmission
        carrier: power
        constraints:
            e_eff: 0.85
        costs:
            monetary:
                e_cap: 200
                om_var: 0.002
```

`ac_transmission` has an efficiency of 0.85, so a loss during transmission of 0.15, as well as some cost definitions.

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). The following figure illustrates this for the example model's transmission technology:

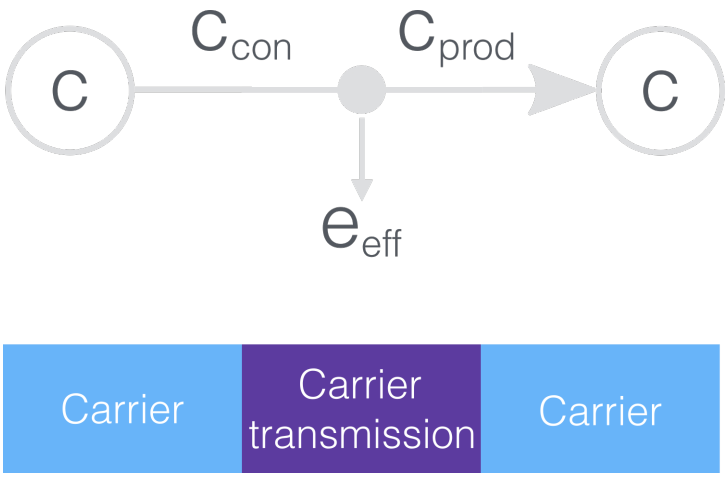


Fig. 1.6: A simple transmission node with an e_{eff}.

Locations

In order to translate the model requirements shown in this section's introduction into a model definition, five locations are used: `r1`, `r2`, `csp1`, `csp2`, and `csp3`.

The technologies are set up in these locations as follows:

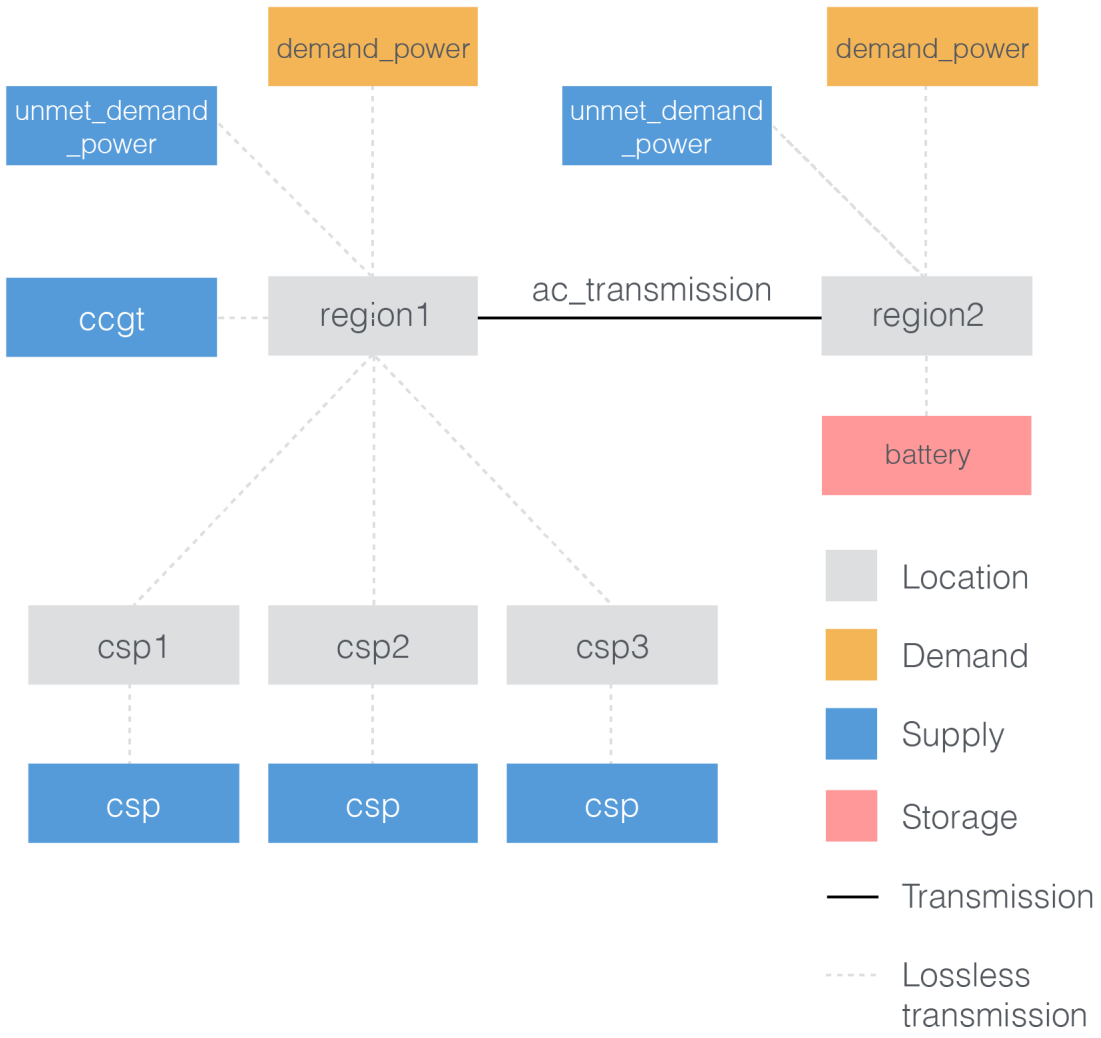


Fig. 1.7: Locations and their technologies in the example model

Let's now look at the first location definition:

```
locations:
    region1:
        techs: ['demand_power', 'unmet_demand_power', 'ccgt']
        override:
            demand_power:
                x_map: 'region1: demand'
                constraints:
                    r: file=demand-1.csv
                    r_scale_to_peak: -40000
            ccgt:
                constraints:
                    e_cap.max: 30000 # increased to ensure no unmet_demand in first␣
↪timestep
```

There are several things to note here:

- The location specifies a list of technologies that it allows (`techs`). 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 `r` 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 demand is then scaled such that the demand peak is at the given value. Note that in Calliope, a supply is positive and a demand is negative, so the peak demand is actually a negative value. Finally, the `x_map` option allows us to read a CSV file with a single column named "demand" and tell Calliope to load data from that column for region `r1`. This is necessary unless the column name(s) in the CSV file already correspond to the location names defined in the model configuration.

The remaining location definitions look like this:

```
    region2:
        techs: ['demand_power', 'unmet_demand_power', 'battery']
        override:
            demand_power:
                x_map: 'region2: demand'
                constraints:
                    r: file=demand-2.csv
                    r_scale_to_peak: -5000

    region1-1,region1-2,region1-3:
        within: region1
        techs: ['csp']
```

`r2` is very similar to `r1`, except that it does not allow the `ccgt` technology. The three `csp` locations are defined together, i.e. they each get the exact same configuration. They are `within` the location `r1` and allow only the `csp` technology, this allows us to model three possible sites for CSP plants within `r1`.

Locations that do not specify a `within` are implicitly at the topmost level. Transmission between locations at the topmost level can only take place if transmission links are defined between them. On the other hand, locations which are specified as `within` another location can automatically and without any losses transmit energy to and from their parent location. In other words, a topmost location and all its contained locations together are implicitly assumed to be on a "copperplate" together. That means there are no transmission constraints and no transmission losses between these locations. Balancing of supply and demand takes place only at the topmost level.

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

```
links:
    region1,region2:
        ac_transmission:
            constraints:
                e_cap.max: 10000
```

1.4.2 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:

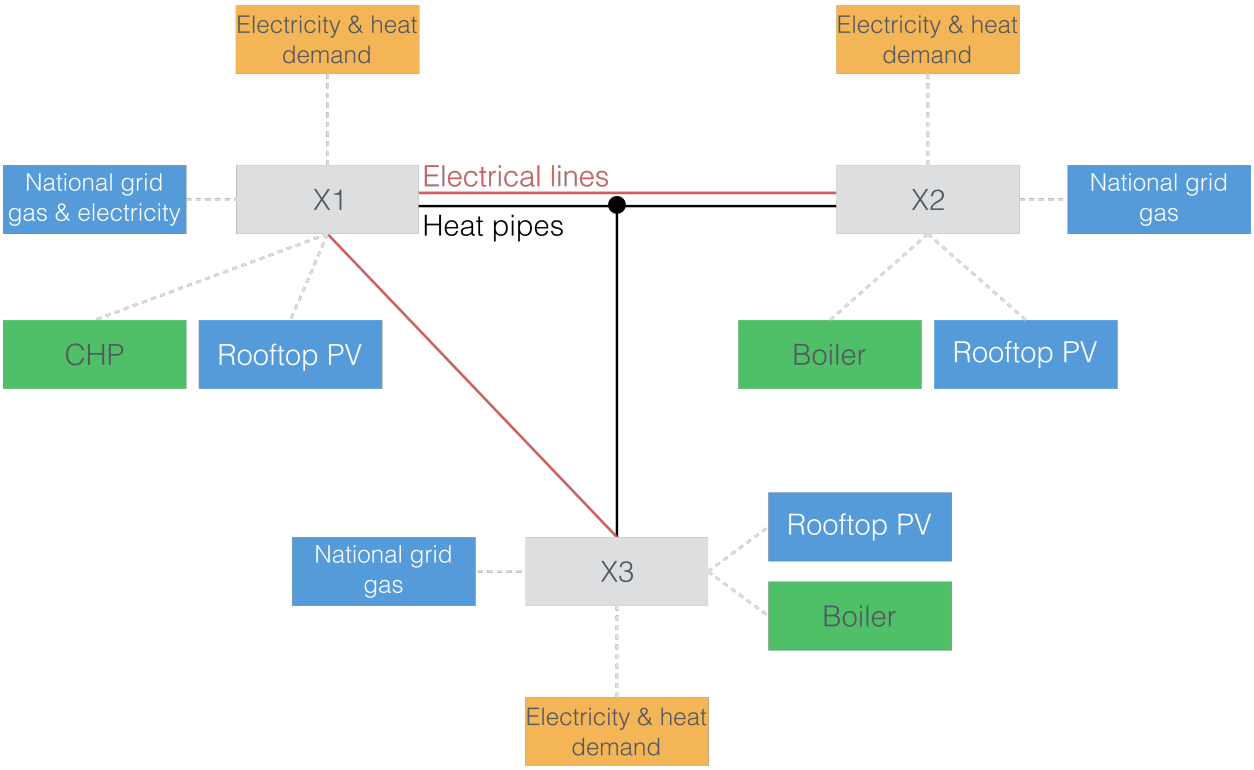


Fig. 1.8: Overview of the built-in urban-scale example model

Supply technologies

This example model defines three supply technologies.

The first two are `national_gas` and `national_grid`, referring to the supply of `gas` (natural gas) and `power` (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.

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

```
##-GRID SUPPLY-##

    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 # 10p/kWh electricity price #ppt

    supply_gas:
        name: 'Natural gas import'
        parent: supply
        carrier: gas
        constraints:
```

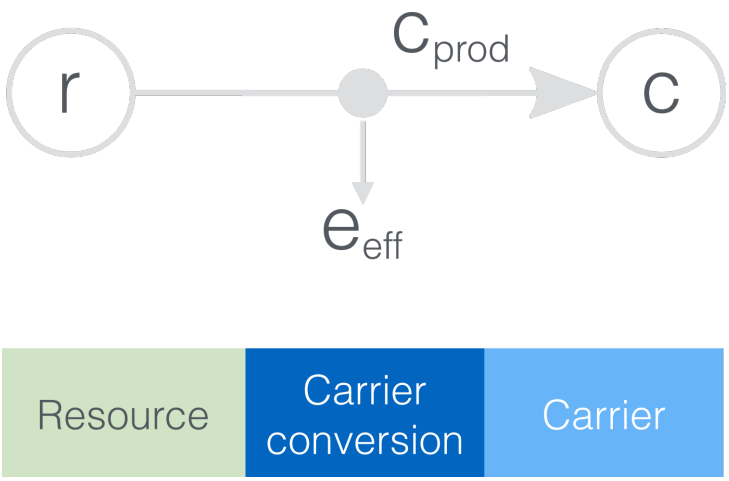


Fig. 1.9: The layout of a simple node, in this case `boiler`, which has one carrier input, one carrier output, a carrier conversion efficiency (e_{eff}), and a constraint on its maximum built e_{cap} (which puts an upper limit on e_{prod}).

```
            r: inf
            e_cap.max: 2000
        costs:
            monetary:
                e_cap: 1
                om_fuel: 0.025 # 2.5p/kWh gas price #ppt
```

The final supply technology is `pv` (solar photovoltaic power), which serves as a inflexible supply technology. It is simple to define, other than having a time-dependant resource availablity, loaded from file. Additionally, it is constrained by available area, which is the rooftop area of the locations in this example.

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

```
##-Renewables-##

    pv:
        name: 'Solar photovoltaic power'
        color: '#99CB48'
        stack_weight: 100
        parent: supply
        export: true
        carrier_out: power
        constraints:
            r: file  # Will look for `pv_r.csv` in data directory - already accounted
↪for panel efficiency
            e_eff: 0.85
            e_cap.max: 250
            r_area.max: 1500
        costs:
            monetary:
                e_cap: 1350
```

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

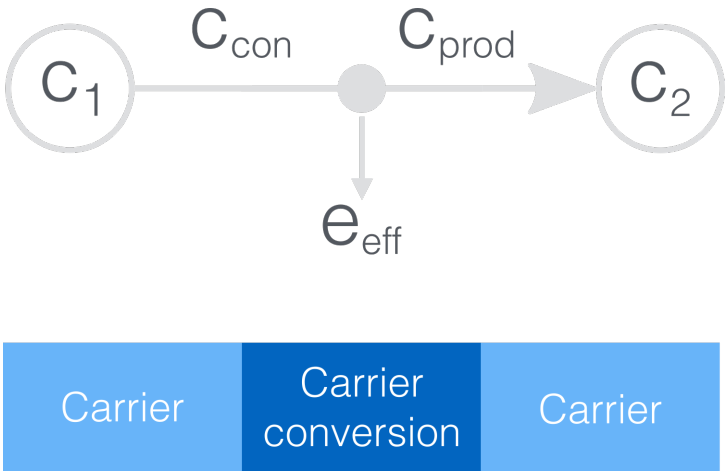


Fig. 1.10: The layout of a simple node, in this case `boiler`, which has one carrier input, one carrier output, a carrier conversion efficiency (e_{eff}), and a constraint on its maximum built e_{cap} (which puts an upper limit on e_{prod}).

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

```
# Conversion

    boiler:
        name: 'Natural gas boiler'
        stack_weight: 100
        parent: conversion
        carrier_out: heat
        carrier_in: gas
        constraints:
            e_cap.max: 600
            e_eff: 0.85
```

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 gasto 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.

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

```
# Conversion_plus

    chp:
        name: 'Combined heat and power'
        stack_weight: 100
        parent: conversion_plus
        export: true
        primary_carrier: power
        carrier_in: gas
```

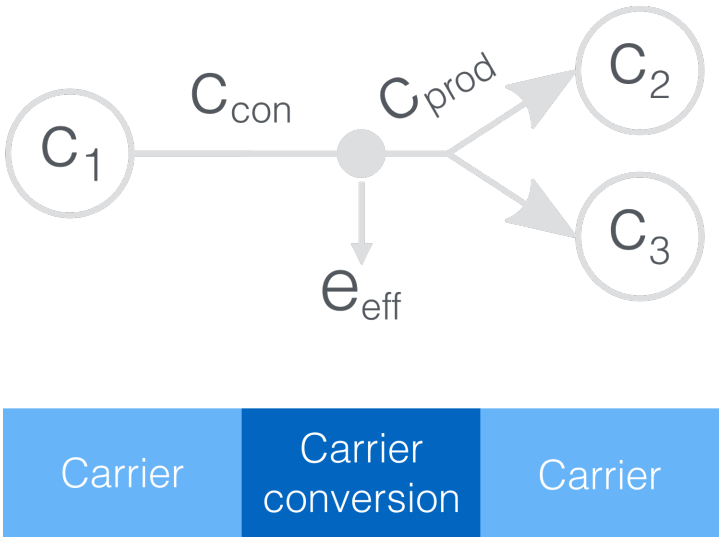


Fig. 1.11: The layout of a more complex node, in this case `chp`, which makes use of multiple output carriers.

```
        carrier_out: power
        carrier_out_2:
            heat: 0.8
        constraints:
            e_cap.max: 1300
            e_eff: 0.405
        costs:
            monetary:
                e_cap: 750
                om_var: 0.004 # .4p/kWh for 4500 operating hours/year
```

Again, `chp` has the definitions for name, color, stack_weight, parent, and carrier_in. Its constraints are no more numerous: it still only defines a carrier conversion efficiency and maximum carrier conversion capacity.

Demand technologies

Electricity and heat demand, and their unmet_demand counterparts are defined here:

```
##-DEMAND-##

    demand_power:
        name: 'Electrical demand'
        parent: demand
        carrier: power

    unmet_demand_power:
        name: 'Unmet electrical demand'
        parent: unmet_demand
        carrier: power

    demand_heat:
        name: 'Heat demand'
        parent: demand
        carrier: heat
```

```
    unmet_demand_heat:
        name: 'Unmet heat demand'
        parent: unmet_demand
```

Electricity and heat demand are a technologies like any other. We will associate an actual demand time series with each demand technology later. The parent of `unmet_demand_power` and `unmet_demand_heat`, `unmet_demand`, is a special kind of supply technology with an unlimited resource but very high cost. It allows a model to remain mathematically feasible even if insufficient supply is available to meet demand, and model results can easily be examined to verify whether there was any unmet demand. There is no requirement to include such a technology in a model, but it is useful to do so, since in its absence, an infeasible model would cause the solver to end with an error, returning no results for Calliope to analyze.

Transmission technologies

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

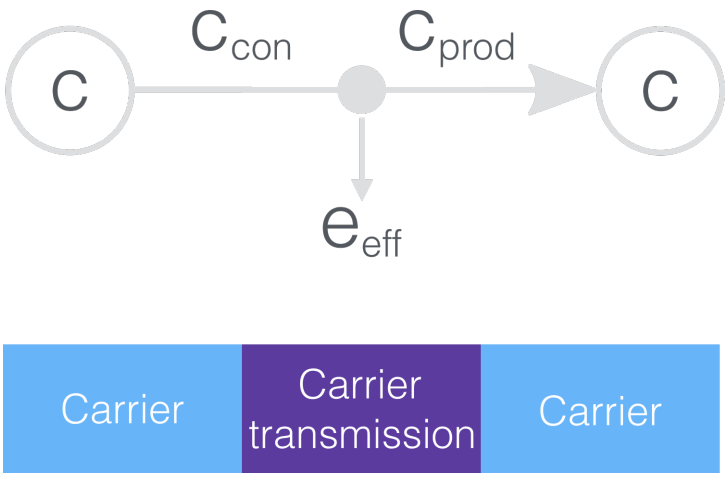


Fig. 1.12: A simple transmission node with an e_{eff}.

```
##-DISTRIBUTION-##

    power_lines:
        name: 'Electrical power distribution'
        parent: transmission
        carrier: power
        constraints:
            e_cap.max: 2000
            e_eff: 0.98
        costs_per_distance:
            monetary:
                e_cap: 0.01

    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:
```

`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%/km. Over the distance between the two locations of 0.5km, this translates to 1.25% 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:

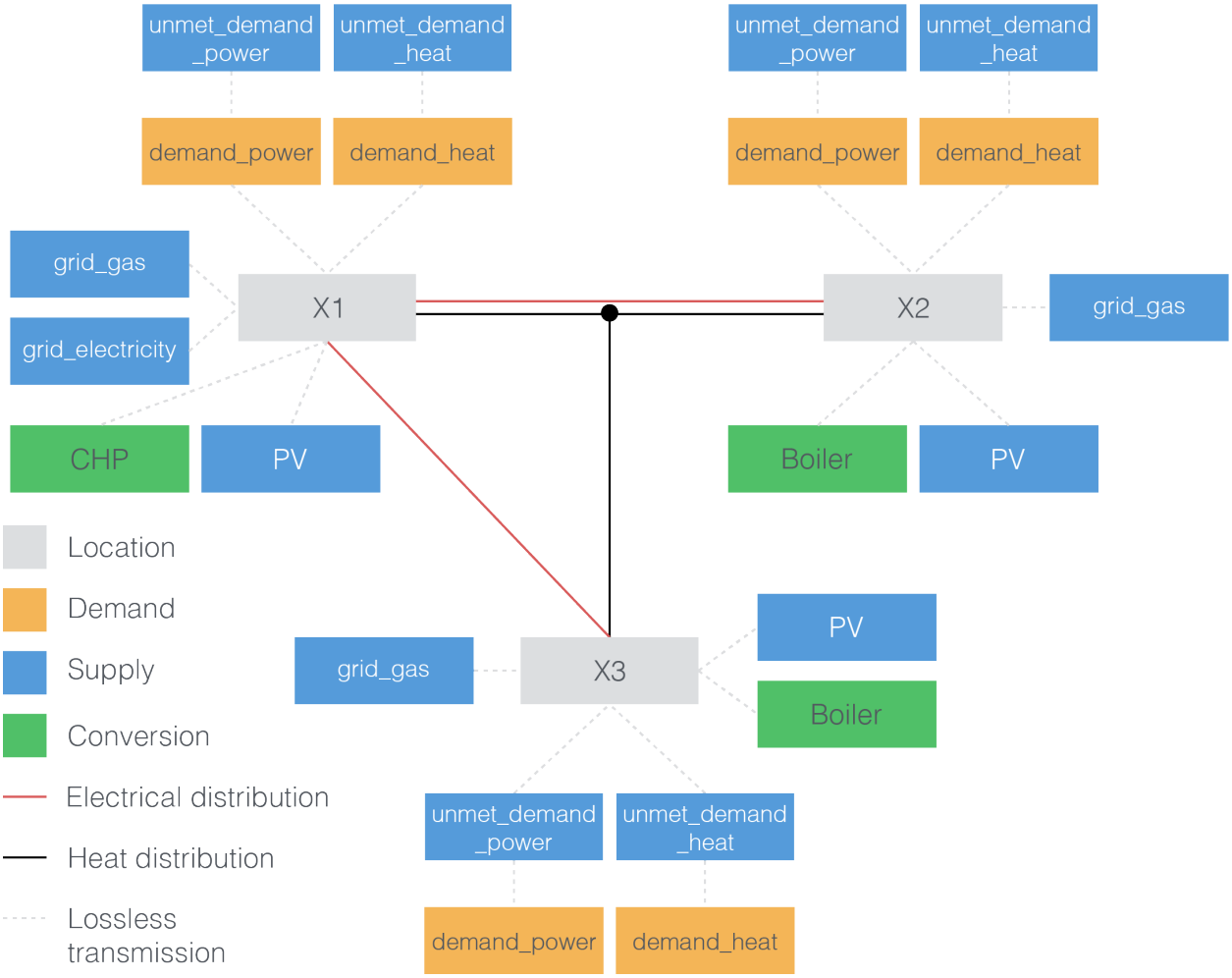


Fig. 1.13: Locations and their technologies in the urban-scale example model

Let's now look at the first location definition:

```
locations:
    X1:
```

```
        techs: ['chp', 'pv',
         'supply_grid_power', 'supply_gas',
         'demand_power', 'demand_heat',
         'unmet_demand_power', 'unmet_demand_heat']
        available_area: 500
        override:
            demand_power.constraints.r: file=demand_power.csv
            demand_heat.constraints.r: file=demand_heat.csv
            supply_grid_power.costs.monetary.e_cap: 100 # cost of transformers
```

There are several things to note here:

- The location specifies a list of technologies that it allows (`techs`). Note that technologies listed here must have been defined elsewhere in the model configuration.
- It also overrides some options for `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 `r` 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 demand is then scaled such that the demand peak is at the given value. Note that in Calliope, a supply is positive and a demand is negative, so the peak demand is actually a negative value. Finally, the `x_map` option allows us to read a CSV file with a single column named "demand" and tell Calliope to load data from that column for region `r1`. This is necessary unless the column name(s) in the CSV file already correspond to the location names defined in the model configuration.

The remaining location definitions look like this:

```
    X2:
        techs: ['boiler', 'pv',
         'supply_gas',
         'demand_power', 'demand_heat',
         'unmet_demand_power', 'unmet_demand_heat'
         ]
        available_area: 1300
        override:
            demand_power.constraints.r: file=demand_power.csv
            demand_heat.constraints.r: file=demand_heat.csv
            boiler.costs.monetary.e_cap: 43.1 # different boiler costs
            pv.costs.monetary:
                        om_var: -0.0203 # revenue for just producing electricity
                        export: -0.0491 # FIT return for PV export

    X3:
        techs: ['boiler', 'pv',
         'supply_gas',
         'demand_power', 'demand_heat',
         'unmet_demand_power', 'unmet_demand_heat'
         ]
        available_area: 900
        override:
            demand_power.constraints.r: file=demand_power.csv
            demand_heat.constraints.r: file=demand_heat.csv
            boiler.costs.monetary.e_cap: 78 # different boiler costs
            pv:
                constraints:
                    e_cap.max: 50 # changing tariff structure below 50kW
                costs.monetary:
```

```
                        om_fixed: -80.5 # reimbursement per kWp from FIT


    N1: # location for branching heat transmission network
        techs: ['heat_pipes']
```

X2 and X3 are very similar to X1, except that they do not connect to the national grid, nor do they contain the chp technology.

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
        techs: ['heat_pipes']
```

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

```
links:
    X1,X2:
        power_lines:
            distance: 10
    X1,X3:
        power_lines:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
    X1,N1:
        heat_pipes:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
    N1,X2:
        heat_pipes:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
    N1,X3:
        heat_pipes:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
```

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 power from the system, in exchange for negative cost (i.e. revenue). To allow this, export: true 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.

1.4.3 Files that define the model

For all Calliope models, including the examples discussed above, the model definitions in through YAML files, which are simple human-readable text files (YAML is a human readable data serialization format). They are stored with a `.yaml` (or `.yml`) extension. See *YAML configuration file format* for details.

Typically, we want to collect all files belonging to a model inside a model directory. In the national-scale example describe above, the layout of that directory, which also includes the time series data in CSV format, is as follows (+ denotes directories, – files):

```
+ example_model
   + model_config
      + data
         - csp_r.csv
         - demand-1.csv
         - demand-2.csv
         - set_t.csv
      - locations.yaml
      - model.yaml
      - techs.yaml
   - run.yaml
```

The urban-scale example follows a similar layout. A complete listing of the files in all example models is available in *Built-in example models*.

Inside the `data` directory, time series are stored as CSV files (their location is configured inside `model.yaml`). At a minimum, a model must always have a `set_t.csv` file which defines the model's timesteps. For more details on this and on time series data more generally, refer to *Using time series data*.

The three files `locations.yaml`, `model.yaml`, and `techs.yaml` together are the model definition, and have been described above. There is one more YAML file, however: `run.yaml`. This tells Calliope *how* to run the model given by the model definition, and will be described next. To run a model in Calliope, these two basic components – a model definition and a run configuration – are always required.

1.4.4 The run configuration

At its most basic, the run configuration simply specifies which model to run, which mode to run it in, and what solver to use. These three options are the required minimum. In the case of the example models, we also specify some output options. The output options only apply when the `calliope run` command-line tool is used to run the model (see below). In the national-scale example:

```
name: "Test run"  # Run name -- distinct from model name!

model: 'model_config/model.yaml'

output:  # Only used if run via the 'calliope run' command-line tool
    format: csv  # Choices: netcdf, csv
    path: 'Output'  # Will be created if it doesn't exist

mode: plan  # Choices: plan, operate

solver: glpk
```

To speed up model runs, the national-scale example model's run configuration also specifies a time subset:

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

The included time series is hourly for a full year. The `subset_t` setting runs the model over only a subset of five days.

The full `run.yaml` file includes additional options, none of which are relevant for this tutorial. See the *full file listing* for the national-scale example and the *section on the run configuration* for more details on the available options.

Plan vs. operate

A Calliope model can either be run in planning mode (`mode: plan`) or operational mode (`mode: operate`). In planning mode, an optimization problem is solved to design an energy system that satisfies the given constraints.

In operational mode, all `max` constraints (such as `e_cap.max`) are treated as fixed rather than as upper bounds. The resulting, fully defined energy system is then operated with a receding horizon control approach. The results are returned in exactly the same format as for planning mode results.

To specify a runnable operational model, capacities for all technologies at all locations would have to be defined. This can be done by specifying `e_cap.equals`. In the absence of `e_cap.equals`, `e_cap.max` is assumed to be fixed.

In this tutorial section, we are only demonstrating the planning mode.

1.4.5 Running a model and analyzing results

Running interactively

The most straightforward way to run a Calliope model is to do so in an interactive Python session.

An example which also demonstrates some of the analysis possibilities after running a model is given in the following Jupyter notebook, based on the national-scale example model. Note that you can download and run this notebook on your own machine (if both Calliope and the Jupyter Notebook are installed):

Calliope interactive national-scale example notebook

Running with the command-line tool

Another way to run a Calliope model is to use the command-line tool `calliope run`. First, we create a new copy of the built-in national-scale example model, by using `calliope new`:

```
$ calliope new testmodel
```

Note: 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=UrbanScale`.

This creates a new directory, `testmodel`, in the current working directory. We can now run this model:

```
$ calliope run testmodel/run.yaml
```

Because of the output options set in `run.yaml`, model results will be stored as a set of CSV files in the directory `Output`. Saving CSV files is an easy way to get results in a format suitable for further processing with other tools. In order to make use of Calliope's analysis functionality, results should be saved as a single NetCDF file instead, which comes with improved performance and handling.

See *Running the model* for more on how to run a model and then retrieve results from it. See *Analyzing results* for more details on analyzing results, including the built-in functionality to read results from either CSV or NetCDF files, making them available for further analysis as described above (*Running interactively*).

1.5 Model 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.

1.5.1 Time-varying vs. constant model parameters

Some model parameters which are defined over the set of time steps `t` can either given as time series or as constant values. If given as constant values, the same value is used for each time step `t`. For details on how to define a parameter as time-varying and how to load time series data into it, see the *time series description in the model configuration section*.

1.5.2 Decision variables

Capacity

- `s_cap(y, x)`: installed storage capacity. Supply plus/Storage only
- `r_cap(y, x)`: installed resource <-> storage conversion capacity
- `e_cap(y, x)`: installed storage <-> grid conversion capacity (gross)
- `r2_cap(y, x)`: installed secondary resource conversion capacity
- `r_area(y, x)`: resource collector area

Unit Commitment

- `r(y, x, t)`: resource <-> storage/carrier_in (+ production, - consumption)
- `r2(y, x, t)`: secondary resource -> storage (+ production)
- `c_prod(c, y, x, t)`: resource/storage/carrier_in -> carrier_out (+ production)
- `c_con(c, y, x, t)`: resource/storage/carrier_in <- carrier_out (- consumption)
- `s(y, x, t)`: total energy stored in device
- `export(y, x, t)`: carrier_out -> export

Costs

- `cost(y, x, k)`: total costs
- `cost_fixed(y, x, k)`: fixed operation costs
- `cost_var(y, x, k, t)`: variable operation costs

1.5.3 Objective function (cost minimization)

Provided by: *calliope.constraints.objective.objective_cost_minimization()*

The default objective function minimizes cost:

$$min: z = \sum_{y}(weight(y) \times \sum_{x} cost(y, x, k = k_m))$$

where k_m is the monetary cost class.

Alternative objective functions can be used by setting the `objective` in the model configuration (see *Model-wide settings*).

weight(y) is 1 by default, but can be adjusted to change the relative weighting of costs of different technologies in the objective, by setting `weight` on any technology (see *Technology*).

1.5.4 Basic constraints

Node resource

Provided by: `calliope.constraints.base.node_resource()`

Defines constraint c_r_available:

$$r_{avail}(y, x, t) = resource(y, x, t) \times r_{scale}(y, x) \times r_{area}(y, x)$$

Which limits the resource flow **to** `supply` and `supply_plus` technologies, or **from** `demand` technologies.

For `supply`:

If the option `constraints.force_r` is set to true, then

$$\frac{c_{prod}(c, y, x, t)}{e_{eff}(y, x, t)} = r_{avail}(y, x, t)$$

If that option is not set:

$$\frac{c_{prod}(c, y, x, t)}{e_{eff}(y, x, t)} \leq r_{avail}(y, x, t)$$

For `demand`:

If the option `constraints.force_r` is set to true, then

$$c_{con}(c, y, x, t) \times e_{eff}(y, x, t) = r_{avail}(y, x, t)$$

If that option is not set:

$$c_{con}(c, y, x, t) \times e_{eff}(y, x, t) \geq r_{avail}(y, x, t)$$

For `supply_plus`:

If the option `constraints.force_r` is set to true, then

$$r(y, x, t) = r_{avail}(y, x, t) \times r_{eff}(y, x, t)$$

If that option is not set:

$$r(y, x, t) \leq r_{avail}(y, x, t) \times r_{eff}(y, x, t)$$

Note: For all other technology types, defining a resource is irrelevant, so they are not constrained here.

Node energy balance

Provided by: `calliope.constraints.base.node_energy_balance()`

Defines nine constraints, which are discussed in turn:

- `c_balance_transmission`: energy balance for `transmission` technologies
- `c_balance_conversion`: energy balance for `conversion` technologies
- `c_balance_conversion_plus`: energy balance for `conversion_plus` technologies
- `c_balance_conversion_plus_secondary_out`: energy balance for `conversion_plus` technologies which have a secondary output carriers
- `c_balance_conversion_plus_tertiary_out`: energy balance for `conversion_plus` technologies which have a tertiary output carriers
- `c_balance_conversion_plus_secondary_in`: energy balance for `conversion_plus` technologies which have a secondary input carriers
- `c_balance_conversion_plus_tertiary_in`: energy balance for `conversion_plus` technologies which have a tertiary input carriers
- `c_balance_supply_plus`: energy balance for `supply_plus` technologies
- `c_balance_storage`: energy balance for `storage` technologies

Transmission balance

Transmission technologies are internally expanded into two technologies per transmission link, of the form `technology_name:destination`.

For example, if the technology `hvdc` is defined and connects `region_1` to `region_2`, the framework will internally create a technology called `hvdc:region_2` which exists in `region_1` to connect it to `region_2`, and a technology called `hvdc:region_1` which exists in `region_2` to connect it to `region_1`.

The balancing for transmission technologies is given by

$$c_{prod}(c, y, x, t) = -1 \times c_{con}(c, y_{remote}, x_{remote}, t) \times e_{eff}(y, x, t) \times e_{eff,perdistance}(y, x)$$

Here, x_{remote}, y_{remote} are x and y at the remote end of the transmission technology. For example, for `(y, x) = ('hvdc:region_2', 'region_1')`, the remotes would be `('hvdc:region_1', 'region_2')`.

$c_{prod}(c, y, x, t)$ for `c='power', y='hvdc:region_2', x='region_1'` would be the import of power from `region_2` to `region_1`, via a `hvdc` connection, at time `t`.

This also shows that transmission technologies can have both a static or time-dependent efficiency (line loss), $e_{eff}(y, x, t)$, and a distance-dependent efficiency, $e_{eff,perdistance}(y, x)$.

For more detail on distance-dependent configuration see *Model configuration*.

Conversion balance

The conversion balance is given by

$$c_{prod}(c_{out}, y, x, t) = -1 \times c_{con}(c_{in}, y, x, t) \times e_{eff}(y, x, t)$$

The principle is similar to that of the transmission balance. The production of carrier c_{out} (the `carrier_out` option set for the conversion technology) is driven by the consumption of carrier c_{in} (the `carrier_in` option set for the conversion technology).

Conversion_plus balance

Conversion plus technologies can have several carriers in and several carriers out, leading to a more complex production/consumption balance.

For the primary carrier(s), the balance is:

$$\sum_{c_{out_1}} \frac{c_{prod}(c_{out_1}, y, x, t)}{carrier_{fraction}(c_{out_1})} = -1 \times \sum_{c_{in_1}} c_{con}(c_{in_1}, y, x, t) \times carrier_{fraction}(c_{in_1}) \times e_{eff}(x, y, t)$$

Where `c_{out_1}` and `c_{in_1}` are the sets of primary production and consumption carriers, respectively and `carrier_{fraction}` is the relative contribution of these carriers, as defined in ??.

The remaining constraints (`c_balance_conversion_plus_secondary_out`, `c_balance_conversion_plus_tertiary_out`, `c_balance_conversion_plus_secondary_in`, `c_balance_conversion_plus_tertiary_in`) link the input/output of the technology secondary and tertiary carriers to the primary consumption/production.

For production:

$$\sum_{c_{out_1}} \frac{c_{prod}}{\frac{(c_{out_1}, y, x, t)}{carrier_{fraction}(c_{out_1})}} \times min(carrier_{fraction}(c_{out_x})) = \sum_{c_{out_x}} c_{prod}(c_{out_x}, y, x, t) \times \frac{carrier_{fraction}(c_{out_x})}{min(carrier_{fraction}(c_{out_x}))}$$

For consumption:

$$\sum_{c_{in_1}} \frac{c_{con}(c_{in_1}, y, x, t)}{carrier_{fraction}(c_{in_1})} \times min(carrier_{fraction}(c_{in_x})) = \sum_{c_{in_x}} c_{con}(c_{in_x}, y, x, t) \times \frac{carrier_{fraction}(c_{in_x})}{min(carrier_{fraction}(c_{in_x}))}$$

Where `x` is either 2 (secondary carriers) or 3 (tertiary carriers).

Warning: The `conversion_plus` technology is still experimental and may not cover all edge cases as intended. Please raise an issue on GitHub if you see unexpected behavior. It is also possible to use a combination of several regular `conversion` technologies to achieve some of the behaviors covered by `conversion_plus`, but at the expense of model complexity.

Supply_plus balance

`Supply_plus` technologies are `supply` technologies with more control over resource flow. You can have multiple resources, a resource capacity, and storage of resource before it is converted to the primary carrier_out.

If storage is possible:

$$s(y, x, t) = s_{minusone} + r(y, x, t) + r_2(y, x, t) - c_{prod}$$

Otherwise:

$$r(y, x, t) = c_{prod} - r_2$$

Where:

c_{prod} is defined as $\frac{c_{prod}(c, y, x, t)}{total_{eff}}$.

$total_{eff}(y, x, t)$ is defined as $e_{eff}(y, x, t) + p_{eff}(y, x, t)$, the plant efficiency including parasitic losses

$r_2(y, x, t)$ is the secondary resource and is always set to zero unless the technology explicitly defines a secondary resource.

$s(y, x, t)$ is the storage level at time t.

$s_{minusone}$ describes the state of storage at the previous timestep. $s_{minusone} = s_{init}(y, x)$ at time $t = 0$. Else,

$$s_{minusone} = (1 - s_{loss}) \times timeres(t - 1) \times s(y, x, t - 1)$$

Note: In operation mode, `s_init` is carried over from the previous optimization period.

Storage balance

Storage technologies balance energy charge, energy discharge, and energy stored:

$$s(y, x, t) = s_{minusone} - c_{prod} - c_{con}$$

Where:

c_{prod} is defined as $\frac{c_{prod}(c,y,x,t)}{total_{eff}}$ if $total_{eff} > 0$, otherwise $c_{prod} = 0$

c_{con} is defined as $c_{con}(c, y, x, t) \times total_{eff}$

$total_{eff}(y, x, t)$ is defined as $e_{eff}(y, x, t) + p_{eff}(y, x, t)$, the plant efficiency including parasitic losses

$s(y, x, t)$ is the storage level at time t.

$s_{minusone}$ describes the state of storage at the previous timestep. $s_{minusone} = s_{init}(y, x)$ at time $t = 0$. Else,

$$s_{minusone} = (1 - s_{loss}) \times timeres(t - 1) \times s(y, x, t - 1)$$

Note: In operation mode, `s_init` is carried over from the previous optimization period.

Node build constraints

Provided by: `calliope.constraints.base.node_constraints_build()`

Built capacity is managed by six constraints.

c_s_cap

This constrains the built storage capacity by:

$$s_{cap}(y, x) \leq s_{cap,max}(y, x)$$

If `y.constraints.s_cap.equals` is set for location `x` or the model is running in operational mode, the inequality in the equation above is turned into an equality constraint.

If both $e_{cap,max}(y, x)$ and $charge\_rate$ are not given, $s_{cap}(y, x)$ is automatically set to zero.

If `y.constraints.s_time.max` is true at location `x`, then `y.constraints.s_time.max` and `y.constraints.e_cap.max` are used to to compute `s_cap.max`. The minimum value of `s_cap.max` is taken, based on analysis of all possible time sets which meet the s_time.max value. This allows time-varying efficiency, $e_{eff}(y, x, t)$ to be accounted for.

c_r_cap

This constrains the built resource conversion capacity by:

$$r_{cap}(y,x) \leq r_{cap,max}(y,x)$$

If the model is running in operational mode, the inequality in the equation above is turned into an equality constraint.

c_r_area

This constrains the resource conversion area by:

$$r_{area}(y,x) \leq r_{area,max}(y,x)$$

By default, `y.constraints.r_area.max` is set to false, and in that case, $r_{area}(y,x)$ is forced to 1.0. If the model is running in operational mode, the inequality in the equation above is turned into an equality constraint. Finally, if `y.constraints.r_area_per_e_cap` is given, then the equation $r_{area}(y,x) = e_{cap}(y,x) * r\_area\_per\_cap$ applies instead.

c_e_cap

This constrains the carrier conversion capacity by:

$$e_{cap}(y,x) \leq e_{cap,max}(y,x) \times e\_cap\_scale$$

If a technology `y` is not allowed at a location `x`, $e_{cap}(y,x) = 0$ is forced.

`y.constraints.e_cap_scale` defaults to 1.0 but can be set on a per-technology, per-location basis if necessary.

If `y.constraints.e_cap.equals` is set for location `x` or the model is running in operational mode, the inequality in the equation above is turned into an equality constraint.

c_e_cap_storage

This constrains the carrier conversion capacity for storage technologies by:

$$e_{cap}(y,x) \leq e_{cap,max}$$

Where $e_{cap,max} = s_{cap}(y,x) * charge\_rate * e\_cap\_scale$

`y.constraints.e_cap_scale` defaults to 1.0 but can be set on a per-technology, per-location basis if necessary.

c_r2_cap

This manages the secondary resource conversion capacity by:

$$r2_{cap}(y,x) \leq r2_{cap,max}(y,x)$$

If `y.constraints.r2_cap.equals` is set for location `x` or the model is running in operational mode, the inequality in the equation above is turned into an equality constraint.

There is an additional relevant option, `y.constraints.r2_cap_follows`, which can be overridden on a per-location basis. It can be set either to `r_cap` or `e_cap`, and if set, sets `c_r2_cap` to track one of these, ie, $r2_{cap,max} = r_{cap}(y,x)$ (analogously for `e_cap`), and also turns the constraint into an equality constraint.

Node operational constraints

Provided by: `calliope.constraints.base.node_constraints_operational()`

This component ensures that nodes remain within their operational limits, by constraining `r`, `c_prod`, `c_con`, `s`, `r2`, and `export`.

r

$r(y, x, t)$ is constrained to remain within $r_{cap}(y, x)$, with the constraint `c_r_max_upper`:

$$r(y, x, t) \leq time\_res(t) \times r_{cap}(y, x)$$

c_prod

$c_prod(c, y, x, t)$ is constrained by `c_prod_max` and `c_prod_min`:

$$c_{prod}(c, y, x, t) \leq time\_res(t) \times e_{cap}(y, x) \times p_{eff}(y, x, t)$$

if `c` is the `carrier_out` of `y`, else $c_{prod}(c, y, x, y) = 0$.

If `e_cap_min_use` is defined, the minimum output is constrained by:

$$c_{prod}(c, y, x, t) \geq time\_res(t) \times e_{cap}(y, x) \times e_{cap,minuse} \times p_{eff}(y, x, t)$$

These contraints are skipped for `conversion_plus` technologies if `c` is not the primary carrier.

c_con

For technologies which are not `supply` or `supply_plus`, $c_con(c, y, x, t)$ is non-zero. Thus $c_con(c, y, x, t)$ is constrainted by `c_con_max`:

$$c_{con}(c, y, x, t) \geq -1 \times timeres(t) \times e_{cap}(y, x)$$

and $c_{con}(c, y, x, t) = 0$ otherwise.

This constraint is skipped for a `conversion_plus` and `conversion` technologies If `c` is a possible consumption carrier (primary, secondary, or tertiary).

s

The constraint `c_s_max` ensures that storage cannot exceed its maximum size by

$$s(y, x, t) \leq s_{cap}(y, x)$$

r2

`c_r2_max` constrains the secondary resource by

$$r2(y, x, t) \leq timeres(t) \times r2_{cap}(y, x)$$

There is an additional check if `y.constraints.r2_startup_only` is true. In this case, $r2(y, x, t) = 0$ unless the current timestep is still within the startup time set in the `startup_time_bounds` model-wide setting. This can be useful to prevent undesired edge effects from occurring in the model.

export

`c_export_max` constrains the export of a produced carrier by

$$export(y, x, t) \leq export_{cap}(y, x)$$

Transmission constraints

Provided by: *calliope.constraints.base.node_constraints_transmission()*

This component provides a single constraint, `c_transmission_capacity`, which forces e_{cap} to be symmetric for transmission nodes. For example, for for a given transmission line between x_1 and x_2, using the technology `hvdc`:

$$e_{cap}(hvdc : x_2, x_1) = e_{cap}(hvdc : x_1, x_2)$$

Node costs

Provided by: `calliope.constraints.base.node_costs()`

These equations compute costs per node.

Weights are adjusted for individual timesteps depending on the timestep reduction methods applied (see *Time resolution adjustment*), and are given by $W(t)$ when computing costs.

The depreciation rate for each cost class `k` is calculated as

$$d(y, k) = \frac{1}{plant\_life(y)}$$

if the interest rate i is 0, else

$$d(y, k) = \frac{i \times (1 + i(y, k))^{plant\_life(k)}}{(1 + i(y, k))^{plant\_life(k)} - 1}$$

Costs are split into fixed and variable costs. The total costs are computed in `c_cost` by

$$cost(y, x, k) = cost_{fixed}(y, x, k) + \sum_t cost_{var}(y, x, k, t)$$

The fixed costs include construction costs, annual operation and maintenance (O&M) costs, and O&M costs which are a fraction of the construction costs. The total fixed costs are computed in `c_cost_fixed` by

$$cost_{fixed}(y, x, k) = cost_{con} + cost_{om,frac} \times cost_{con} + cost_{om,fixed} \times e_{cap}(y, x) \times \frac{\sum_t timeres(t) \times W(t)}{8760}$$

Where

$$\begin{aligned} cost_{con} = d(y, k) \times & \frac{\sum_t timeres(t) \times W(t)}{8760} \\ & \times (cost_{s\_cap}(y, k) \times s_{cap}(y, x) \\ & + cost_{r\_cap}(y, k) \times r_{cap}(y, x) \\ & + cost_{r\_area}(y, k) \times r_{area}(y, x) \\ & + cost_{e\_cap}(y, k) \times e_{cap}(y, x)) \\ & + cost_{r2\_cap}(y, k) \times r2_{cap}(y, x)) \end{aligned}$$

The costs are as defined in the model definition, e.g. $cost_{r\_cap}(y, k)$ corresponds to `y.costs.k.r_cap`.

For transmission technologies, $cost_{e\_cap}(y,k)$ is computed differently, to include the per-distance costs:

$$cost_{e\_cap,transmission}(y,k) = \frac{cost_{e\_cap}(y,k) + cost_{e\_cap,perdistance}(y,k)}{2}$$

This implies that for transmission technologies, the cost of construction is split equally across the two locations connected by the technology.

The variable costs are O&M costs applied at each time step:

$$cost_{var} = cost_{op,var} + cost_{op,fuel} + cost_{op,r2} + cost_{op,export}$$

Where:

$$cost_{op,var} = cost_{om\_var}(k,y,x,t) \times \sum_t W(t) \times c_{prod}(c,y,x,t)$$

$$cost_{op,fuel} = \frac{cost_{om\_fuel}(k,y,x,t) \times \sum_t W(t) \times r(y,x,t)}{r_{eff}(y,x)}$$

$$cost_{op,r2} = \frac{cost_{om\_r2}(k,y,x,t) \times \sum_t W(t) \times r_2(y,x,t)}{r2_{eff}(y,x)}$$

$$cost_{op,export} = cost_{export}(k,y,x,t) \times export(y,x,t)$$

If $cost_{om\_fuel}(k,y,x,t)$ is given for a `supply` technology and $e_{eff}(y,x) > 0$ for that technology, then:

$$cost_{op,fuel} = cost_{om\_fuel}(k,y,x,t) \times \sum_t W(t) \times \frac{c_{prod}(c,y,x,t)}{e_{eff}(y,x)}$$

`c` is the technology primary `carrier_out` in all cases.

Model balancing constraints

Provided by: `calliope.constraints.base.model_constraints()`

Model-wide balancing constraints are constructed for nodes that have children:

$$\sum_{y,x \in X_i} c_{prod}(c,y,x,t) + \sum_{y,x \in X_i} c_{con}(c,y,x,t) = 0 \qquad \forall i,t$$

i are the level 0 locations, and X_i is the set of level 1 locations (x) within the given level 0 location, together with that location itself.

There is also the need to ensure that technologies cannot export more energy than they produce:

$$c_{prod}(c,y,x,t) \geq export(y,x,t)$$

1.5.5 Planning constraints

These constraints are loaded automatically, but only when running in planning mode.

System margin

Provided by: *`calliope.constraints.planning.system_margin()`*

This is a simplified capacity margin constraint, requiring the capacity to supply a given carrier in the time step with the highest demand for that carrier to be above the demand in that timestep by at least the given fraction:

$$\sum_{y}\sum_{x} c_{prod}(c,y,x,t_{max,c}) \times (1+m_c) \leq timeres(t) \times \sum_{y_c}\sum_{x}(e_{cap}(y,x)/e_{eff}(y,x,t_{max,c}))$$

where y_c is the subset of `y` that delivers the carrier `c` and m_c is the system margin for that carrier.

For each carrier (with the name `carrier_name`), Calliope attempts to read the model-wide option `system_margin.carrier_name`, only applying this constraint if a setting exists.

System-wide capacity

Provided by: *calliope.constraints.planning.node_constraints_build_total()*

This constraint sets a maximum for capacity, `e_cap`, across all locations for any given technology:

$$\sum_{x} e_{cap}(x,y) \leq e_{cap,total\_max}(y)$$

If $e_{cap,total\_equals}$ is given instead, this becomes $\sum_x e_{cap}(x,y) \leq e_{cap,total\_max}(y)$.

$$\sum_{y}\sum_{x} c_{prod}(c,y,x,t_{max,c}) \times (1+m_c) \leq timeres(t) \times \sum_{y_c}\sum_{x}(e_{cap}(y,x)/e_{eff}(y,x,t_{max,c}))$$

where y_c is the subset of `y` that delivers the carrier `c` and m_c is the system margin for that carrier.

For each carrier (with the name `carrier_name`), Calliope attempts to read the model-wide option `system_margin.carrier_name`, only applying this constraint if a setting exists.

1.5.6 Optional constraints

Optional constraints are included with Calliope but not loaded by default (see the *configuration section* for instructions on how to load them in a model).

These optional constraints can be used both in planning and operational modes.

Ramping

Provided by: *calliope.constraints.optional.ramping_rate()*

Constrains the rate at which plants can adjust their output, for technologies that define `constraints.e_ramping`:

$$diff = \frac{c_{prod}(c,y,x,t) + c_{con}(c,y,x,t)}{timeres(t)} - \frac{c_{prod}(c,y,x,t-1) + c_{con}(c,y,x,t-1)}{timeres(t-1)}$$

$$max\_ramping\_rate = e_{ramping} \times e_{cap}(y,x)$$

$$diff \leq max\_ramping\_rate$$

$$diff \geq -1 \times max\_ramping\_rate$$

Group fractions

Provided by: *calliope.constraints.optional.group_fraction()*

This component provides the ability to constrain groups of technologies to provide a certain fraction of total output, a certain fraction of total capacity, or a certain fraction of peak power demand. See *Parents and groups* in the configuration section for further details on how to set up groups of technologies.

The settings for the group fraction constraints are read from the model-wide configuration, in a `group_fraction` setting, as follows:

```
group_fraction:
   capacity:
      renewables: ['>=', 0.8]
```

This is a minimal example that forces at least 80% of the installed capacity to be renewables. To activate the output group constraint, the `output` setting underneath `group_fraction` can be set in the same way, or `demand_power_peak` to activate the fraction of peak power demand group constraint.

For the above example, the `c_group_fraction_capacity` constraint sets up an equation of the form

$$\sum_{y^*}\sum_{x} e_{cap}(y,x) \geq fraction \times \sum_{y}\sum_{x} e_{cap}(y,x)$$

Here, y^* is the subset of y given by the specified group, in this example, `renewables`. $fraction$ is the fraction specified, in this example, 0.8. The relation between the right-hand side and the left-hand side, $\geq$, is determined by the setting given, >=, which can be ==, <=, or >=.

If more than one group is listed under `capacity`, several analogous constraints are set up.

Similarly, `c_group_fraction_output` sets up constraints in the form of

$$\sum_{y^*}\sum_{x}\sum_{t} c_{prod}(c,y,x,t) \geq fraction \times \sum_{y}\sum_{x}\sum_{t} c_{prod}(c,y,x,t)$$

Finally, `c_group_fraction_demand_power_peak` sets up constraints in the form of

$$\sum_{y^*}\sum_{x} e_{cap}(y,x) \geq fraction \times (-1 - m_c) \times peak$$

$$peak = \frac{\sum_x r(y_d,x,t_{peak}) \times r_{scale}(y_d,x)}{timeres(t_{peak})}$$

This assumes the existence of a technology, `demand_power`, which defines a demand (negative resource). y_d is `demand_power`. m_c is the capacity margin defined for the carrier `c` in the model-wide settings (see *System margin*). t_{peak} is the timestep where $r(y_d,x,t)$ is maximal.

Whether any of these equations are equalities, greater-or-equal-than inequalities, or lesser-or-equal-than inequalities, is determined by whether >=, <=, or == is given in their respective settings.

Available area

Provided by: *calliope.constraints.optional.max_r_area_per_loc()*

Where several technologies require space to acquire resource (e.g. solar collecting technologies) at a given location, this constraint provides the ability to limit the total area available at a location:

$$area_{available}(x) \geq \sum_{y}\sum_{xi} r_{area}(y,xi)$$

Where `xi` is the set of locations within the family tree, descending from and including `x`.

1.6 Model configuration

Note: See *Configuration reference* for a complete listing of all available configuration options.

To run a model, two things are needed: a *model definition* that defines such things as technologies, locations, costs and constraints, and *run settings*, which specify how the given model should be run. At their most basic, these two components can be specified in just two YAML files:

- `model.yaml`, which sets up the model and may import any number of additional files in order to split large models up into manageable units. It must also specify, via the `data_path` setting, the directory with data files for those technologies that have data explicit in space and time. The data directory must contain, at a minimum, a file called `set_t.csv` which defines the model's timesteps. See *Using time series data* below for more information on this.
- `run.yaml`, which sets up run-specific and environment-specific settings such as which solver to use. It must also, with the `model` setting, specify which model should be run, by pointing to that model's primary model configuration file (e.g., `model.yaml`).

Either of these files can have an arbitrary name, but it makes sense to call them something like `run.yaml` (for the run settings) and `model.yaml` (for the model definition).

The remainder of this section deals with the model configuration. See *Run configuration* for the run configuration.

The model definition can be split into several files in two ways:

1. Model configuration files can can use an `import` statement to 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. If a setting is defined both in the importing file and the imported file, the imported settings are overridden.
2. The `model` setting in the run settings may either give a single file or a list of files, which will be combined on model initialization. An example of this is:

```
model:
    - model.yaml  # Define general model settings
    - techs.yaml   # Define technologies, their constraints and costs
    - locations.yaml  # Define locations and transmission capacities
```

Note: Calliope includes a command-line tool, `calliope new`, which will create a new model at the given path, based on the built-in national-scale example model and its run configuration:

```
calliope new my_new_model
```

This makes it easier to experiment with the built-in example, and to quickly create a model by working off an existing skeleton.

1.6.1 Technologies

A technology's identifier can be any alphanumeric string. The index of all technologies `y` is constructed at model instantiation from all defined technologies. At the very minimum, a technology should define some constraints and some costs. A typical supply technology that has an infinite resource without spatial or temporal definition might define:

```
my_tech:
   parent: 'supply'
   name: 'My test technology'
```

```
   carrier_out: 'some_energy_carrier'
   constraints:
      e_cap.max: 1000  # kW
   costs:
      monetary:
         e_cap: 500  # per kW of e_cap.max
```

A demand technology, with its demand data stored in a time series in the file `demand.csv`, might look like this:

```
my_demand_tech:
   parent: 'demand'
   carrier_in: 'some_energy_carrier'
   constraints:
      r: 'file=demand.csv'
```

Technologies must always define a parent, and this can either be one of the pre-defined abstract base technologies or another previously defined technology. The pre-defined abstract base technologies that can be inherited from 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.
- `unmet_demand`: Supplies unlimited energy to a carrier with a very high cost, but does not get counted as a supply technology for analysis and grouping purposes. An `unmet_demand` technology for all relevant carriers should usually be included in a model to keep the solution feasible in all cases (see the *tutorials* for a practical example).
- `unmet_demand_as_supply_tech`: Works like `unmet_demand` but is a normal `supply` technology, so it does get counted as a supply technology for analysis and grouping purposes.
- `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 specifies (which, in turn, inherits from its own parent). The abstract base technologies inherit from a model-wide default technology called `defaults`.

It is possible, for example, to define a `wind` technology that specifies generic characteristics for wind power plants, and then multiple additional technologies, such as `wind_onshore` and `wind_offshore`, that specify `parent: wind`, but also override some of the generic wind settings with their own.

See *Overriding technology options* below for additional information on how technology settings propagate through the model and how they can be overridden.

Refer to *Technology* for a complete list of all available technology constraints and costs.

Note: The identifiers of the abstract base technologies are reserved and cannot be used for a user-defined technology. In addition, `defaults` is also a reserved identifier and cannot be used.

Parents and groups

Because each technology must define a `parent`, the definition of all technologies represents a tree structure, with the built-in defaults representing the root node, the built-in abstract base technologies inheriting from that root node, and all other user-defined technologies inheriting from one of the abstract base technologies.

There are two important aspects to this model definition structure.

First, only leaf nodes (the outermost nodes) in this tree may actually be used as technologies in model definitions. In other words, the parent-child inheritance structure allows technologies to inherit settings from their parents, but only those technologies without any children themselves are considered "real". Calliope will raise an error if this requirement is not met.

Second, every non-leaf node is implicitly a group of technologies, and the solution returned by Calliope reports aggregated information for each defined technology and its children (see *Analyzing results*).

The `group` option only has an effect on supply diversity functionality in the analysis module (again, see *Analyzing results* for details). Because every non-leaf technology is implicitly a group, those that should be considered as distinct groups for the purpose of diversity of supply must be explicitly marked with `group: true`.

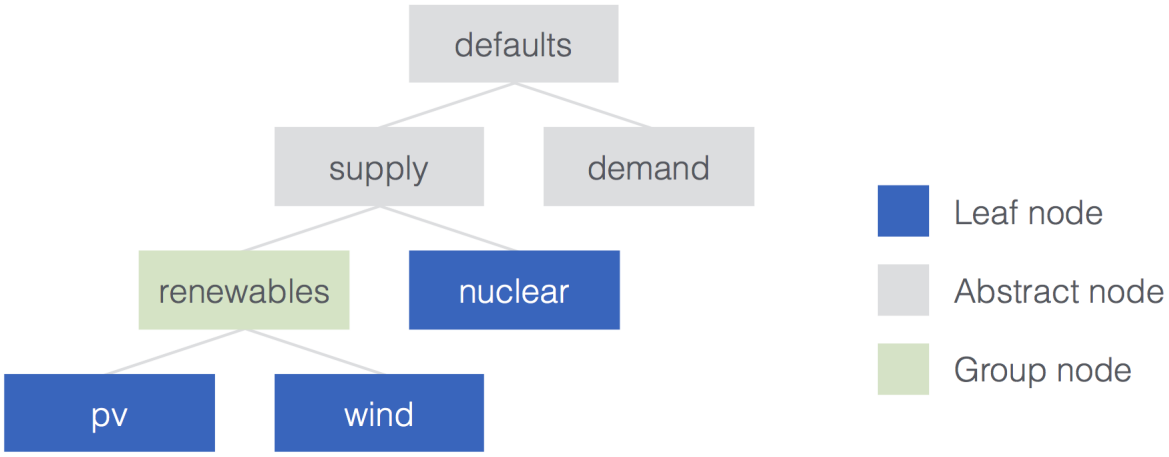


Fig. 1.14: An example of a simple technology inheritance tree. `renewables` could define any defaults that both `pv` and `wind` should inherit, furthermore, it sets `group: true`. Thus, for purposes of supply diversity, `pv` and `wind` will be counted together, while `nuclear` will be counted separately.

1.6.2 Locations

A location's name can be any alphanumeric string, but using integers makes it easier to define constraints for a whole range of locations by using the syntax `from--to`. Locations can be given as a single location (e.g., `location1`), a range of integer location names using the `--` operator (e.g., `0--10`), or a comma-separated list of alphanumeric location names (e.g., `location1,location2,10,11,12`). Using `override`, some settings can be overridden on a per-location and per-technology basis (see below).

Locations may also define a parent location using `within`, as shown in the following example:

```
locations:
    location1:
        techs: ['demand_power', 'nuclear']
```

```
        override:
            nuclear:
                constraints:
                    e_cap.max: 10000
    location2:
        techs: ['demand_power']
    offshore1, offshore2:
        within: location2
        techs: ['offshore_wind']
```

The energy balancing constraint looks at a location's level to decide which locations to consider in balancing supply and demand. Locations that are not `within` another location are implicitly at the topmost level. Supply and demand within locations on the topmost level must always be be balanced, but they can exchange energy with each other via transmission technologies, which may define parameters such as costs, distance, and losses.

Locations that are contained within a parent location have implicit loss-free and cost-free transmission between themselves and the parent location. The balancing constraint makes sure that supply and demand within a location and its direct children is balanced.

Warning: If a location contained within a parent location itself defines children, it is no longer included in the implicit free transmission between its siblings and parent location. In turn, it receives implicit free transmission with its own children.

Transmission nodes

A location can also act as just a branch in a transmission network. This is relevant for locations where transmission links split into several lines, without any other technologies at those locations. In this case, the location definition becomes:

```
locations:
      location1:
          techs: ['transmission-tech']
```

Where `transmission-tech` can refer to any previously defined `transmission` technology which passes through that location. Listing transmission technologies is not necessary for any other location type.

1.6.3 Transmission links

Transmission links are defined in the model definition as follows:

```
links:
   location1,location2:
      transmission-tech:
         constraints:
            e_cap.max: 10000
   location1,location3:
      transmission-tech:
         # ...
      another-transmission-tech:
         # ...
```

`transmission-tech` can refer to any previously defined technology, but that technology must have the abstract base technology `transmission` as a parent

It is possible to specify multiple possible transmission technologies (e.g., with different costs or efficiencies) between two locations by simply listing them all.

Transmission links can also specify a distance, which transmission technologies can use to compute distance-dependent costs or efficiencies. An `e_loss` can be specified under `constraints_per_distance` and costs for any cost class can be specified under `costs_per_distance` (see example below).

```
links:
   location1,location2:
      transmission-tech:
         distance: 500

techs:
   transmission-tech:
      # per_distance constraints specified per 100 units of distance
      per_distance: 100
      constraints_per_distance:
         e_loss: 0.01  # loss per 100 units of distance
      costs_per_distance:
         monetary:
            e_cap: 10  # cost per 100 units of distance
```

Note: 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, in 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`.

1.6.4 Overriding technology options

Technologies can define generic options, for example `name`, constraints, for example `constraints.e_cap.max`, and costs, for example `costs.monetary.e_cap`.

These options can be overridden in several ways, and whenever such an option is accessed by Calliope it works its way through the following list until it finds a definition (so entries further up in this list take precedence over those further down):

1. Override for a specific location `x1` and technology `y1`, which may be defined via `locations` (e.g. `locations.x1.override.y1.constraints.e_cap.max`)
2. Setting specific to the technology `y1` if defined in `techs` (e.g. `techs.y1.constraints.e_cap.max`)
3. Check whether the immediate parent of the technology `y` defines the option (assuming that `y1` specifies `parent: my_parent_tech`, e.g. `techs.my_parent_tech.constraints.e_cap.max`)
4. If the option is still not found, continue along the chain of parent-child relationships. Since every technology should inherit from one of the abstract base technologies, and those in turn inherit from the model-wide defaults, this will ultimately lead to the model-wide default setting if it has not been specified anywhere else. See *Technology constraints* for a complete listing of those defaults.

The following technology options can be overriden on a per-location basis:

- `x_map`
- `constraints.*`
- `constraints_per_distance.*`
- `costs.*`

The following settings cannot be overridden on a per-location basis:

- Any other options, such as `parent` or `carrier`
- `costs_per_distance.*`
- `depreciation.*`

1.6.5 Using 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 overrides, 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).

Defining a model's time steps

Irrespective of whether it actually uses time-varying parameters, a model must at specify its timesteps with a file called `set_t.csv`. This must contain two columns (comma-separated), the first one being integer indices, and the second, ISO 8601 compatible timestamps (usually in the format `YYYY-MM-DD hh:mm:ss`, e.g. `2005-01-01 00:00:00`).

For example, the first few lines of a file specifying hourly timesteps for the year 2005 would look like this:

```
0,2005-01-01 00:00:00
1,2005-01-01 01:00:00
2,2005-01-01 02:00:00
3,2005-01-01 03:00:00
4,2005-01-01 04:00:00
5,2005-01-01 05:00:00
6,2005-01-01 06:00:00
```

Defining time-varying parameters

For parameters that vary in time, time series data can be read from CSV files. This can be done in two ways (using the example of `r`):

1. Specify `r: file=filename.csv` to pick the desired CSV file.
2. Specify `r: file`. In this case, the file name is automatically determined according to the format `tech_param.csv` (e.g., `pv_r.csv` for the parameter `r` of a technology with the identifier `pv`).

Each CSV file must have integer indices in the first column which match the integer indices from `set_t.csv`. The first row must be column names, while the rest of the cells are the actual (integer or floating point) data values:

```
,loc1,loc2,loc3,...
0,10,20,10.0,...
1,11,19,9.9,...
2,12,18,9.8,...
...
```

In the most straightforward case, the column names in the CSV files correspond to the location names defined in the model (in the above example, `loc1`, `loc2` and `loc3`). However, it is possible to define a mapping of column names to locations. For example, if our model has two locations, `uk` and `germany`, but the electricity demand data columns are `loc1`, `loc2` and `loc3`, then the following `x_map` definition will read the demand data for the desired locations from the specified columns:

```
electricity_demand:
   x_map: 'uk: loc1, germany: loc2'
   constraints:
      r: 'file=demand.csv'
```

Warning: After reading a CSV file, if any columns are missing (i.e. if a file does not contain columns for all locations defined in the current model), the value for those locations is simply set to 0 for all timesteps.

Note: `x_map` maps column names in an input CSV file to locations defined in the model, in the format `name_in_model: name_in_file`, with as many comma-separated such definitions as necessary.

In all cases, all CSV files, alongside `set_t.csv`, must be inside the data directory specified by `data_path` in the model definition.

For example, the files for a model specified in `model.yaml`, which defined `data_path: model_data`, might look like this (+ are directories, – files):

```
- model.yaml
+ model_data/
   - set_t.csv
   - tech1_r.csv
   - tech2_r.csv
   - tech2_e_eff.csv
   - ...
```

When reading time series, the `r_scale_to_peak` option can be useful. Specifying this will automatically scale the time series so that the peak matches the given value. In the case of `r` for demand technologies, where `r` will be negative, the peak is instead a trough, and this is handled automatically. In the below example, the electricity demand timeseries is loaded from `demand.csv` and scaled such that the demand peak is 60,000:

```
electricity_demand:
   constraints:
      r: 'file=demand.csv'
      r_scale_to_peak: -60000
```

Calliope provides functionality to automatically adjust the resolution of time series data to make models more computationally tractable. See *Time resolution adjustment* for details on this.

1.6.6 Loading optional constraints

Calliope uses "constraint generator" functions that read the model configuration and build model constraints based on it. Constraint generators for optional constraints are included in the *calliope.constraints.optional* module. In addition, custom-built user constraints can be added by loading additional constraint generator functions. They can be added in `model.yaml` by specifying `constraints`, for example:

```
constraints:
    - constraints.optional.ramping_rate
    - my_custom_module.my_constraint
```

When resolving constraint names, Calliope first checks whether the constraint is part of Calliope itself (in the above example, this is the case for `constraints.optional.ramping_rate`, which is included in Calliope). If the constraint is not found as part of Calliope, the first part of the dot-separated name is interpreted as a Python module name (in the above example, `my_custom_module`). The module is imported and the constraint loaded from it.

This architecture makes it possible to add constraints in a modular way without modifying the Calliope source code. Custom constraints have access to all model configuration, so that additional settings can easily be included anywhere in the model configuration to support the functionality of custom constraints. See *Development guide* for information on this.

1.7 Run configuration

Note: See *Run settings* in the configuration reference for a complete listing of all available configuration options.

At a minimum, the run configuration must provide three settings, as shown in this example:

```
model: 'model_config/model.yaml'
mode: 'plan'
solver: 'glpk'
```

`model` specifies the path to the model configuration file for the model to be run. `mode` specifies whether the model should be run in planning (`plan`) or operational (`operate`) mode (see *Running the model*). Finally, `solver` specifies the solver to be used. Calliope has been tested with GLPK, Gurobi and CPLEX. Any of the solvers that Pyomo is compatible with should work.

Additional (optional) settings, including debug settings, can be specified in the run configuration. In particular, the run settings can override any model settings by specifying `override`, e.g.:

```
override:
   techs:
      nuclear:
         costs:
            monetary:
               e_cap: 1000
```

Note: If run settings override the `data_path` setting and specify a relative path, that path will be interpreted as relative to the run settings file and not the model settings file being overridden.

Instead of directly overriding settings within the run configuration file using an `override` block, it is also possible to specify an additional model configuration file with overriding settings by using the `model_override: path/to/model_override.yaml` setting (the path given here is relative to the run configuration file).

The optional settings to adjust the timestep resolution and those for parallel runs are discussed below. For a complete list of the other available settings, see *Run settings* in the configuration reference.

1.7.1 Time resolution adjustment

Models must have a default timestep length (defined implicitly by the timesteps defined in `set_t.csv`), and all time series files used in a given model must conform to that timestep length requirement.

However, this default resolution can be adjusted over parts of the dataset via configuring `time` in the run settings. At its most basic, this allows running a function that can perform arbitrary adjustments to the time series data in the model, via `time.function`, and/or applying a series of masks to the time series data to select specific areas of interest, such as periods of maximum or minimum production by certain technologies, via `time.masks`.

The available options include:

1. Uniform time resolution reduction through the resample function, which takes a pandas-compatible rule describing the target resolution. For example, to resample to 6-hourly timesteps:

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

2. Deriving representative days from the input time series, by applying either k-means or hierarchical clustering as defined in *calliope.time_clustering*, for example:

```
time:
    function: apply_clustering
    function_options: {clustering_func: 'get_clusters_kmeans', how: 'mean', k: 20}
```

3. Heuristic selection: application of one or more of the masks defined in *calliope.time_masks*, via a list of masks given in `time.masks`. See *Time series* in the API documentation for the available masking functions. 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), as in this example which looks for the week of minimum and maximum potential wind production (assuming a `wind` technology was specified), then reduces the rest of the input time series to 6-hourly resolution:

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

Note: When loading a model, all time steps initially have the same weight. Time step resolution reduction methods may adjust the weight of individual timesteps; this is 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. See the implementation of constraints in *calliope.constraints.base* for more detail.

1.7.2 Settings for parallel runs

The run settings can also include a `parallel` section.

This section is parsed when using the `calliope generate` command-line tool to generate a set of runs to be executed in parallel (see *Parallel runs*). A run settings file defining `parallel` can still be used to execute a single model run, in which case the `parallel` section is simply ignored.

The concept behind parallel runs is to specify a base model (via the run configuration's `model` setting), then define a set of model runs using this base model, but overriding one or a small number of settings in each run. For example, one could explore a range of costs of a specific technology and how this affects the result.

Specifying these iterations is not (yet) automated, they must be manually entered under `parallel.iterations:` section. However, Calliope provides functionality to gather and process the results from a set of parallel runs (see *Analyzing results*).

At a minimum, the `parallel` block must define:

- a `name` for the run
- the `environment` of the cluster (if it is to be run on a cluster), currently supported is `bsub` and `qsub`. In either case, the generated scripts can also be run manually
- `iterations`: a list of model runs, with each entry giving the settings that should be overridden for that run. The settings are *run settings*, so, for example, `time.function` can be overridden. Because the run settings can themselves override model settings, via `override`, model settings can be specified here, e.g. `override.techs.nuclear.costs.monetary.e_cap`.

The following example parallel settings show the available options. In this case, two iterations are defined, and each of them overrides the nuclear `e_cap` costs (`override.techs.nuclear.costs.monetary.e_cap`):

```
parallel:
    name: 'example-model'  # Name of this run
    environment: 'bsub'  # Cluster environment, choices: bsub, qsub
    data_path_adjustment: '../../../model_config'
    # Execute additional commands in the run script before starting the model
    pre_run: ['source activate pyomo']
    # Execute additional commands after running the model
    post_run: []
    iterations:
        - override.techs.nuclear.costs.monetary.e_cap: 1000
        - override.techs.nuclear.costs.monetary.e_cap: 2000
    resources:
        threads: 1  # Set to request a non-default number of threads
        wall_time: 30  # Set to request a non-default run time in minutes
        memory: 1000  # Set to request a non-default amount of memory in MB
```

This also shows the optional settings available:

- `data_path_adjustment`: replaces the `data_path` setting in the model configuration during parallel runs only
- `pre_run` and `post_run`: one or multiple lines (given as a list) that will be executed in the run script before / after running the model. If running on a computing cluster, `pre_run` is likely to include a line or two setting up any environment variables and activating the necessary Python environment.
- `resources`: specifying these will include resource requests to the cluster controller into the generated run scripts. `threads`, `wall_time`, and `memory` are available. Whether and how these actually get processed or honored depends on the setup of the cluster environment.

For an iteration to override more than one setting at a time, the notation is as follows:

```
iterations:
    - first_option: 500
      second_option: 10
```

```
    - first_option: 600
      second_option: 20
```

See *Parallel runs* in the section on running models for details on how to use the parallel settings to generate and execute parallel runs.

1.8 Running the model

There are two basic modes for the model: planning mode and operational mode. The mode is set in the run configuration.

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 (see *Model-wide settings* for the settings that control the receding horizon).

In either case, there are three ways to run the model:

1. With the `calliope run` command-line tool.
2. By generating and then executing parallel runs with the `calliope generate` command-line tool.
3. By programmatically creating and running a model from within other Python code, or in an interactive Python session.

1.8.1 Single runs with the command-line tool

The included command-line tool `calliope run` will execute a given run configuration:

```
$ calliope run my_model/run.yaml
```

It will generate and solve the model, then save the results to the the output directory given by `output.path` in the run configuration.

Two output formats are available: a collection CSV files or a single NetCDF file. They can be chosen by settings `output.format` in the run configuration (set to `netcdf` or `csv`). The *`read`* module provides methods to read results stored in either of these formats, so that they can then be analyzed with the *`analysis`* module.

1.8.2 Parallel runs

Warning: This functionality is currently not Windows-compatible.

Scripts to simplify the creation and execution of a large number of Calliope model runs are generated with the `calliope generate` command-line tool as follows:

- Create a `run.yaml` file with a `parallel:` section as needed (see *Settings for parallel runs*).
- On the command line, run `calliope generate path/to/run.yaml`.
- By default, this will create a new subdirectory inside a `runs` directory in the current working directory. You can optionally specify a different target directory by passing a second path to `calliope generate`, e.g. `calliope generate path/to/run.yaml path/to/my_run_files`.

- Calliope generates several files and directories in the target path. The most important are the `Runs` subdirectory which hosts the self-contained configuration for the runs and `run.sh` script, which is responsible for executing each run. In order to execute these runs in parallel on a compute cluster, a submit.sh script is also generated containing job control data, and which can be submitted via a cluster controller (e.g., `qsub submit.sh`).

The `run.sh` script can simply be called with an integer argument from the sequence (1, number of parallel runs) to execute a given run, e.g. `run.sh 1`, `run.sh 2`, etc. This way the runs can easily be executed irrespective of the parallel computing environment available.

Note: Models generated via `calliope generate` automatically save results as a single NetCDF file per run inside the parallel runs' `Output` subdirectory, regardless of whether the `output.path` or `output.format` options have been set.

See *Settings for parallel runs* for details on configuring parallel runs.

1.8.3 Running programmatically from other Python code

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

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

If `config_run` 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 run configuration.

`config_run` can also take an *AttrDict* object containing the configuration. Furthermore, `Model()` has an `override` parameter, which takes an `AttrDict` with settings that will override the given run settings.

After instantiating the `Model` object, and before calling the `run()` method, it is possible to manually inspect and adjust the configuration of the model.

After the model has been solved, an xarray Dataset containing solution variables and aggregated statistics is accessible under the `solution` property on the model instance.

The *API documentation* gives an overview of the available methods for programmatic access.

1.8.4 Extracting results from a completed model run

If running single runs via the command-line tool or using the parallel run functionality, results will be saved as either a single NetCDF file per model run or a set of CSV files per model run. These can then be read back into an interactive Python session for analysis – see *Analyzing results* – or further processed with any other tool available to the modeller.

When working with the in-memory `solution` object, which is an n-dimensional xarray.Dataset, the xarray documentation should be consulted (this will be the case either in interactive runs, or after having read it back into memory from disk),

It is easy to extract 2-dimensional slices from the solution by using xarray's ability to extract pandas DataFrames. See the *Tutorials* for examples of how this is done.

The easiest path to extracting data from a model without dealing with xarray, pandas, or other Python data analysis tools, is to set the `output.format` in the run configuration to `csv`, which results in CSV files that can be read for example with common spreadsheet software.

1.8.5 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 some run configuration options to make it easier to determine the cause of the first two of these possibilities. See the *debugging options described in the configuration reference*.

Python debugging

If using Calliope interactively in a Python session and/or developing custom constraints and analysis functionality, we recommend reading up on the Python debugger and (if using IPython or Jupyter Notebooks) making heavy use of the %debug magic.

1.9 Analyzing results

1.9.1 The solution object

On successfully solving a model, Calliope creates a `solution`, which is a multi-dimensional xarray.Dataset, with the model and run configuration stored as *`AttrDict`* attributes of the dataset (`config_model` and `config_run`).

The analysis tools included with Calliope expect to operate on a dataset.

The solution contains model variables such as `rs`, `s`, `e_cap`, `r_area`, etc, as well as variables derived from them such as `capacity_factor` and `levelized_cost`. It also contains several two-dimensional summary and metadata tables:

- `metadata`: metadata for each technology (such as its `stack_weight` or `color`), used for analysis and plotting.
- `groups`: definition of technology groups and their members.
- `shares`: technology and group based shares of production, consumption and installed capacity (index is `y`).
- `summary`: summary information on each technology.

1.9.2 Reading solutions

Calliope provides functionality to read a solution from a single NetCDF file or a collection of CSV files and reconstruct a `solution` object for further analysis in a Python session:

```
solution_from_netcdf = calliope.read.read_netcdf('my_solution.nc')

solution_from_csv = calliope.read.read_csv('path/to/output_directory')
```

1.9.3 Reading results from parallel runs

A successfully completed parallel run will contain multiple solutions inside its "Output" directory. To read all solutions, including information about the iterations they correspond to, use:

```
results = calliope.read.read_dir('path/to/Output')
```

The `results` variable is an *AttrDict* with two keys:

- `iterations`: a DataFrame containing the iterations from this parallel run
- `solutions`: an AttrDict with iteration IDs as keys and the individual `solution` objects as values

This allows easy access to and analysis of solutions.

1.9.4 Analyzing solutions

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.

Note: The built-in analysis and plotting functionality is still experimental. More documentation on it will be added in a future release.

1.10 Configuration reference

1.10.1 YAML configuration file format

All configuration files (with the exception of time series data files) are in the YAML format, "a human friendly data serialization 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.

Warning: When generating parallel runs with the `calliope generate` command-line tool, any `import` directive, unlike other settings that point to file system paths such as `model_override` or `data_path`, is evaluated immediately and all imported files are combined into one model configuration file for the parallel runs. This means that while paths used in `import` directives don't need adjustment for parallel runs, other settings that work with file system paths probably do need adjustment to account for the way files are laid out on the system running the parallel runs. For this purpose, the `data_path_adjustment` inside a `parallel` configuration block can change the data path for parallel runs only.

1.10.2 Model-wide settings

These settings can either be in a central `model.yaml` file, or imported from other files if desired.

Mandatory model-wide settings with no default values (see *Model configuration* for more information on defining `techs`, `locations` and `links`):

```
data_path: 'data'  # Path to CSV (time series) data files

techs:
    # ... technology definitions ...

locations:
    # ... location definitions ...

links:
    # ... transmission link definitions ...
```

Optional model-wide settings with no default values (example settings are shown here):

```
constraints:  # List of additional constraints
    - constraints.optional.ramping_rate
    # ... other constraints to load ...

group_fraction:
    # ... setup for group_fraction constraints (see model formulation section) ...

metadata:  # Metadata for analysis and plotting
    map_boundary: []
    location_coordinates:
    location_ordering:
```

Optional model-wide settings that have defaults set by Calliope (default values are shown here):

```
# Chooses the objective function
# If not set, defaults to the included cost minimization objective
objective:  'constraints.objective.objective_cost_minimization'
```

```
startup_time: 12  # Length of startup period (hours)

opmode:  # Operation mode settings
    horizon: 48  # Optimization period length (hours)
    window: 24  # Operation period length (hours)

system_margin:  # Per-carrier system margins
    power: 0
```

1.10.3 Technology

A technology with the identifier `tech_identifier` is configured by a YAML block within a `techs` block. The following block shows all available options and their defaults (see further below for the constraints, costs, and depreciation definitions):

```
tech_identifier:
    name:  # A descriptive name, e.g. "Offshore wind"
    parent:  # An abstract base technology, or a previously defined one
    carrier: false # Energy carrier to produce/consume, for all except conversion
    stack_weight: 100  # Weight of this technology in the stack when plotting
    color: false  # HTML color code, or `false` to choose a random color
    group: false  # Make this a group for purposes of supply diversity analysis
    weight: 1.0 # Cost weighting in objective function
    constraints:
        # ... constraint definitions ...
    costs:
        monetary:
            # ... monetary cost definitions ...
        # ... other cost classes ...
    depreciation:
        # ... depreciation definitions ...
```

Each technology **must** define a `parent`, which can either be an abstract base technology such as `supply`, or any other technology previously defined in the model. The technology inherits all settings from its parent, but overwrites anything it specifies again itself. See *Parents and groups* for more details on this and on the function of the `group:` option.

Each technology **must** also define at least one of the `carrier` options. `carrier` implicitly defines `carrier_in` & `carrier_out` for storage and transmission technologies, `carrier_in` for demand technologies, and `carrier_out` for supply/supply_plus technologies. Supply and demand technologies can be defined using `carrier_in`/`carrier_out` instead, which will produce the same result. For conversion and conversion_plus, there are further options available:

```
tech_identifier:
    primary_carrier: false # Setting the primary carrier_out to associate with costs &
↪ constraints, if multiple primary carriers are assigned
    carrier_in: false # Primary energy carrier(s) to consume
    carrier_in_2: false # Secondary energy carrier(s) to consume, conversion_plus only
    carrier_in_3: false # Tertiary energy carrier(s) to consume, conversion_plus only
    carrier_out: false # Primary energy carrier(s) to produce
    carrier_out_2: false # Secondary energy carrier(s) to produce, conversion_plus␣
↪only
    carrier_out_3: false # Tertiary energy carrier(s) to produce, conversion_plus only
```

If carriers are given at secondary or tertiary level, they are given in an indented list, with their consumption/production with respect to `carrier_in`/`carrier_out`. For example:

```
tech_identifier_1:
    carrier_in: 'primary_consumed_carrier'
    carrier_in_2:
        secondary_consumed_carrier: 0.8 # consumes 0.8 units of ``secondary_consumed_
↪carrier`` for every 1 unit of ``primary_consumed_carrier``
    carrier_in_3:
        tertiary_consumed_carrier: 0.1 # consumes 0.1 units of ``tertiary_consumed_
↪carrier`` for every 1 unit of ``primary_consumed_carrier``
    carrier_out: 'primary_produced_carrier'
    carrier_out_2:
        secondary_produced_carrier: 0.5 # produces 0.5 units of ``secondary_produced_
↪carrier`` for every 1 unit of ``primary_produced_carrier``
    carrier_out_3:
        tertiary_produced_carrier: 0.9 # produces 0.9 units of ``tertiary_produced_
↪carrier`` for every 1 unit of ``primary_produced_carrier``
```

Where multiple carriers are included in a carrier level, any of those carriers can meet the carrier level requirement. They are listed in the same indented level, for example:

```
tech_identifier_1:
    primary_carrier: 'primary_produced_carrier' # ``primary_produced_carrier`` will␣
↪be used to cost/constraint application
    carrier_in:
        primary_consumed_carrier: 1 # if chosen, will consume 1 unit of ``primary_
↪consumed_carrier`` to meet the requirements of ``carrier_in``
        primary_consumed_carrier_2: 0.5 # if chosen, will consume 0.5 units of␣
↪``primary_consumed_carrier_2`` to meet the requirements of ``carrier_in``
    carrier_in_2:
        secondary_consumed_carrier: 0.8 # if chosen, will consume 0.8 units of␣
↪``secondary_consumed_carrier`` for every 1 unit of ``carrier_in`` being consumed
        secondary_consumed_carrier_2: 0.1 # if chosen, will consume 0.1 / 0.8 = 0.125␣
↪units of ``secondary_consumed_carrier_2`` for every 1 unit of ``carrier_in`` being␣
↪consumed
    carrier_out:
        primary_produced_carrier: 1 # if chosen, will produce 1 unit of ``primary_
↪produced_carrier`` for every 1 unit of ``carrier_out`` being produced
        primary_produced_carrier_2: 0.8 # if chosen, will produce 0.8 units of␣
↪``primary_produced_carrier_2`` for every 1 unit of ``carrier_in`` being produced
```

Note: 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).

`stack_weight` and `color` determine how the technology is shown in model outputs. The higher the `stack_weight`, the lower a technology will be shown in stackplots.

The `depreciation` definition is optional and only necessary if defaults need to be overridden. However, at least one constraint (such as `e_cap.max`) and one cost should usually be defined.

Transmission technologies can additionally specify per-distance constraints and per-distance costs (see *Transmission links*). Currently, only `e_loss` constraints and `e_cap` costs are supported:

```
transmission_tech:
   # per_distance constraints specified per 100 units of distance
   per_distance: 100
   constraints_per_distance:
```

```
    e_loss: 0.01  # loss per 100 units of distance
  costs_per_distance:
    monetary:
      e_cap: 10  # cost per 100 units of distance
```

Note: Transmission technologies can define both an `e_loss` (per-distance) and an `e_eff` (distance-independent). For example, setting `e_eff` to 0.9 implies a 10% loss during transmission, independent of distance. If both `e_loss` and `e_eff` are defined, their effects are cumulative.

1.10.4 Technology 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	Details
force_r	false	Forces this technology to use all available `r`, rather than making it a maximum upper boundary (for production) or minimum lower boundary (for consumption)
r_unit	power	Sets the unit of `r` to either `power` (i.e. kW) or `energy` (i.e. kWh), which affects how resource time series are processed when performing time resolution adjustments
r_eff	1.0	Resource to/from storage conversion efficiency
r_area.min	0	Minimum installed collector area (m^2)
r_area.max	false	Maximum installed collector area (m^2), set to false by default in order to disable this constraint and force r_area to 1
r_area.equals	false	Specific installed collector area (m^2)
r_area_per_e_cap	false	If set, forces `r_area` to follow `e_cap` with the given numerical ration (e.g. setting to 1.5 means that `r_area == 1.5 * e_cap`)
r_cap.min	0	Minimum installed resource to/from storage conversion capacity (kW)
r_cap.max	inf	Maximum installed resource to/from storage conversion capacity (kW)
r_cap.equals	false	Specific installed resource to/from storage conversion capacity (kW)
r_cap_equals_e_cap	false	If true, r_cap is forced to equal e_cap
r_scale	1.0	Scale resource by this value
r_scale_to_peak	false	Scale resource such that its peak is as defined here, `false` to disable. This setting only has an effect if a time series is used via `r: file`
allow_r2	false	Allow secondary resource
r2_startup_only	false	Allow secondary resource during startup_time only (only has an effect if `allow_r2` is `true`)
r2_eff	1.0	Secondary resource to/from storage conversion efficiency
r2_cap.min	0	Minimum installed secondary resource to storage conversion capacity (kW)
r2_cap.max	inf	Maximum installed secondary resource to storage conversion capacity (kW)
r2_cap.equals	0	Specific installed secondary resource to storage conversion capacity (kW)
r2_cap_follow	false	Can be set to `e_cap` or `r_cap` to set `r2_cap.max` to the respective value (in which case, any given `r2_cap.max` is ignored). `false` to disable
r2_cap_follow_mode	'max'	Can be set to *max* or *equals* to specifiy which `r2_cap` constraint is specific by the variable given in `r2_cap_follow`
s_init	0	Initial storage level (kWh)
s_cap.min	0	Minimum storage capacity (kWh)

Continued on next page

Table 1.1 – continued from previous page

Setting	Default	Details
s_cap.max	0	Maximum storage capacity (kWh). If both this and s_time.max are set to non-zero values, the minimum resulting constraint of either s_time.max or s_cap.max is applied.
s_cap.equals	false	Specific storage capacity (kWh)
c_rate	false	Charge rate (0 to 1) defining maximum charge/discharge (kW) for a given maximum storage capacity (kWh)
s_time.max	0	Max storage time (full load hours). If both this and s_cap.max are set to non-zero values, the minimum resulting constraint of either s_time.max or s_cap.max is applied.
s_loss	0	Storage loss rate (per hour)
e_prod	true	Allow this technology to supply energy to the carrier
e_con	false	Allow this technology to consume energy from the carrier
p_eff	1.0	Plant parasitic efficiency (additional losses as energy gets transferred from the plant to the carrier, e.g. due to plant parasitic consumption)
e_eff	1.0	Storage to/from carrier conversion efficiency. Can be set to `file` or `file:` or to a single numerical value
e_cap.min	0	Minimum installed storage to/from carrier conversion capacity (kW), per location
e_cap.max	0	Maximum installed storage to/from carrier conversion capacity (kW), per location
e_cap.equals	false	Specific installed storage to/from carrier conversion capacity (kW), per location
e_cap.total_max	inf	Maximum installed storage to/from carrier conversion capacity (kW), model-wide
e_cap.total_equals	false	Specific installed storage to/from carrier conversion capacity (kW), model-wide
e_cap_scale	1.0	Scale all `e_cap` min/max/equals/total_max/total_equals constraints by this value
e_cap_min_use	false	Set to a value between 0 and 1 to force minimum storage to carrier capacity use for production technologies
e_ramping	false	Ramping rate (fraction of installed capacity per hour), set to `false` to disable ramping constraints (only has an effect if the optional ramping constraints are loaded)
export_cap	false	Maximum allowed export for a technology, set to `false` to disable.

1.10.5 Technology 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	Details
s_cap	0	Cost of storage capacity (per kWh)
r_area	0	Cost of resource collector area (per m^2)
r_cap	0	Cost of resource conversion capacity (per kW)
r2_cap	0	Cost of secondary resource conversion capacity (per kW)
e_cap	0	Cost of carrier conversion capacity (per kW gross)
om_frac	0	Yearly O&M costs (fraction of total investment)
om_fixed	0	Yearly O&M costs (per kW of `e_cap`)
om_var	0	Variable O&M costs (per kWh of `es_prod`)
om_fuel	0	Fuel costs (per kWh of `r` used)
om_r2	0	Fuel costs for secondary resource (per kWh of `rb` used)
export	0	Cost of exporting excess energy (per kWh of `export`). Usually used in the negative sense, as a subsidy.

1.10.6 Technology depreciation

These technology depreciation settings apply when calculating levelized costs. The interest rate can be set on a per-cost class basis, and defaults to 0.10 for `monetary` and 0 for every other cost class.

```
plant_life: 25  # Lifetime of a plant (years)
interest:
    default: 0  # Default interest rate if not specified for a cost class ``k``
    monetary: 0.10  # Interest rate for the ``monetary`` cost class
```

1.10.7 Abstract base technologies

This lists all pre-defined abstract base technologies and the defaults they provide. Note that it is not possible to define a technology with the same identifier as one of the abstract base technologies. In addition to providing default values for some options, which abstract base technology a user-defined technology inherits from determines how Calliope treats the technology internally. This internal treatment means that only a subset of available constraints are used for each of the abstract base technologies.

supply

```
parent: defaults
constraints:
    r: inf
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints*:

```
stack_weight
color
carrier_out
group
x_map
export
constraints:
    r
    force_r
    r_unit
    r_area.min
```

```
    r_area.max
    r_area.equals
    r_area_per_e_cap
    e_prod
    e_eff
    e_cap.min
    e_cap.max
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
    export_cap
costs:
    r_area
    e_cap
    om_frac
    om_fixed
    om_var
    om_fuel
    export
depreciation:
    plant_life
    interest
weight
```

supply_plus

```
parent: defaults
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints*:

```
stack_weight
color
carrier_out
group
x_map
export
constraints:
    r
    force_r
    r_unit
    r_eff
    r_area.min
    r_area.max
    r_area.equals
    r_area_per_e_cap
    r_cap.min
    r_cap.max
    r_cap.equals
    r_cap_equals_e_cap
    r_scale
    r_scale_to_peak
    allow_r2
    r2_startup_only
```

```
    r2_eff
    r2_cap.min
    r2_cap.max
    r2_cap.equals
    r2_cap_follow
    r2_cap_follow_mode
    s_init
    s_cap.min
    s_cap.max
    s_cap.equals
    c_rate
    s_time.max
    s_loss
    e_prod
    p_eff
    e_eff
    e_cap.min
    e_cap.max
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
    export_cap
costs:
    s_cap
    r_area
    r_cap
    r2_cap
    e_cap
    om_frac
    om_fixed
    om_var
    om_fuel
    om_r2
    export
depreciation:
    plant_life
    interest
weight
```

demand

```
parent: defaults
constraints:
    r: 0
    force_r: true
    e_cap.max: inf
    e_prod: false
    e_con: true
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints*:

```
stack_weight
color
```

```
carrier_in
group
x_map
export
constraints:
    r
    force_r
    r_unit
    r_area.min
    r_area.max
    r_area.equals
    r_area_per_e_cap
    e_con
    e_eff
    e_cap.min
    e_cap.max
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
costs:
    r_area
    e_cap
    om_frac
    om_fixed
    om_var
    export
depreciation:
    plant_life
    interest
weight
```

unmet_demand

```
stack_weight: 0
color: '#666666'
parent: defaults
constraints:
    r: inf
    e_cap.max: inf
costs:
    monetary:
        om_var: 1.0e+9
```

There is also the option to include unmet demand as a "true" supply technology by making use of `unmet_demand_as_supply_tech`:

```
stack_weight: 0
color: '#666666'
parent: supply
constraints:
    e_cap.max: inf
costs:
    monetary:
```

```
        om_var: 1.0e+9
```

In either case, the additional available constraints are the same as found for the supply abstract base technology. However, it is generally not advised to edit any constraints pertaining to *unmet_demand*.

storage

Warning: The default value provided by `storage` for *e_con'* should not be overridden.

```
parent: defaults
constraints:
    e_con: true
    r: inf # not used but has to be defined as infinite to avoid issues
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints* :

```
stack_weight
color
carrier
group
x_map
export
constraints:
    e_prod
    s_init
    s_cap.min
    s_cap.max
    s_cap.equals
    c_rate
    s_time.max
    s_loss
    e_eff
    e_cap.min
    e_cap.max
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
    export_cap
costs:
    s_cap
    e_cap
    om_frac
    om_fixed
    om_var
    export
depreciation:
    plant_life
    interest
weight
```

transmission

Warning: The default value provided by `transmission` for“e_con“ should not be overridden.

```
parent: defaults
per_distance: 1
constraints:
    e_cap.max: inf
    e_con: true
    r: inf # not used but has to be defined as infinite to avoid issues
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints* :

```
stack_weight
color
carrier
group
x_map
export
constraints:
    e_prod
    e_eff
    e_cap.min
    e_cap.max
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
    export_cap
costs:
    e_cap
    om_frac
    om_fixed
    om_var
    export
costs_per_distance:
    e_cap
constraints_per_distance:
    e_loss
depreciation:
    plant_life
    interest
weight
```

conversion

```
parent: defaults
constraints:
    e_con: true
    r: inf # not used but has to be defined as infinite to avoid issues
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints* :

```
stack_weight
color
carrier_in
carrier_out
group
x_map
export
constraints:
    e_prod
    e_eff
    e_cap.min
    e_cap.max
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
    export_cap
costs:
    e_cap
    om_frac
    om_fixed
    om_var
    export
depreciation:
    plant_life
    interest
weight
```

conversion_plus

```
parent: defaults
constraints:
    e_con: true
    r: inf # not used but has to be defined as infinite to avoid issues
```

Available constraints are as follows, with full descriptions found above, in *Technology constraints* :

```
stack_weight
color
primary_carrier
carrier_in
carrier_in_2
carrier_in_3
carrier_out
carrier_out_2
carrier_out_3
group
x_map
export
constraints:
    e_prod
    e_eff
    e_cap.min
    e_cap.max
```

```
    e_cap.equals
    e_cap.total_max
    e_cap.total_equals
    e_cap_scale
    e_cap_min_use
    e_ramping
    export_cap
costs:
    e_cap
    om_frac
    om_fixed
    om_var
    export
depreciation:
    plant_life
    interest
weight
```

1.10.8 Run settings

These settings will usually be in a central `run.yaml` file, which may import from other files if desired.

Mandatory settings:

- `model`: Path to the model configuration which is to be used for this run
- `mode`: `plan` or `operate`, whether to run the model in planning or operational mode
- `solver`: Name of the solver to use

Optional settings:

- **Output options – these are only used when the model is run via the `calliope run` command-line tool:**
  - `output.path`: Path to an output directory to save results (will be created if it doesn't exist already)
  - `output.format`: Format to save results in, either `netcdf` or `csv`
- `parallel`: Settings used to generate parallel runs, see *Settings for parallel runs* for the available options
- `time`: Settings to adjust time resolution, see *Time resolution adjustment* for the available options
- `override`: Override arbitrary settings from the model configuration. E.g., this could specify `techs.nuclear.costs.monetary.e_cap: 1000` to set the `e_cap` costs of `nuclear`, overriding whatever was set in the model configuration
- `model_override`: Path to a YAML configuration file which contains additional overrides for the model configuration. If both this and `override` are specified, anything defined in `override` takes precedence over model configuration added in the `model_override` file.
- `solver_options`: A list of options, which are passed on to the chosen solver, and are therefore solver-dependent (see below)

Debugging failing runs

A number of run settings exist to make debugging failing runs easier:

- `subset_y`, `subset_x`, `subset_t`: specify if only a subset of technologies (y), locations (x), or timesteps (t) should be used for this run. This can be useful for debugging purposes. The timestep subset can be specified as `[startdate, enddate]`, e.g. `['2005-01-01', '2005-01-31']`. The subsets are processed before building the model and applying time resolution adjustments, so time resolution functions will only see the reduced set of data.

In addition, settings relevant to debugging can be specified inside a `debug` block as follows:

- `debug.keep_temp_files`: Whether to keep temporary files inside a `Logs` directory rather than deleting them after completing the model run (which is the default). Useful to debug model problems.
- `debug.overwrite_temp_files`: When `debug.keep_temp_files` is true, and the `Logs` directory already exists, Calliope will stop with an error, but if this setting is true, it will overwrite the existing temporary files.
- `debug.symbolic_solver_labels`: By default, Pyomo uses short random names for all generated model components, rather than the variable and parameter names used in the model setup. This is faster but for debugging purposes models must be human-readable. Thus, particularly when using `debug.keep_temp_files: true`, this setting should also be set to `true`.
- `debug.echo_solver_log`: Displays output from the solver on screen while solving the model (by default, output is only logged to the log file, which is removed unless `debug.keep_temp_files` is true).

The following example debug block would keep temporary files, removing possibly existing files from a previous run beforehand:

```
debug:
    keep_temp_files: true
    overwrite_temp_files: true
```

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:

```
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:

```
solver: cplex

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

1.11 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 can provide a simple working model.

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

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

1.11.1 National-scale example

Available as `calliope.examples.NationalScale`.

Model settings

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

```
+ model_config
   + data
      - csp_r.csv
      - demand-1.csv
      - demand-2.csv
      - set_t.csv
   - locations.yaml
   - model.yaml
   - techs.yaml
```

`model.yaml`:

```
##
# IMPORT OTHER FILES
##

# Can either be paths relative to this file, or absolute paths

import:
    - 'techs.yaml'
    - 'locations.yaml'

##
# MODEL NAME
##

name: "Test model"


##
# DATASET PATH
##

# Can either be a path relative to this file, or an absolute path

data_path: 'data'
```

```
##
# OBJECTIVE FUNCTION
##

# 'constraints.objective.objective_cost_minimization' is used by default
# objective:

##
# ADDITIONAL CONSTRAINTS
##

constraints:
    - constraints.optional.ramping_rate

##
# OTHER MODEL-WIDE OPTIONS
##

system_margin:
    power: 0
    heat: 0
```

techs.yaml:

```
##
# TECHNOLOGY DEFINITIONS
##

techs:

    ##
    # Supply
    ##
    ccgt:
        name: 'Combined cycle gas turbine'
        color: '#FDC97D'
        stack_weight: 200
        parent: supply
        carrier_out: power
        constraints:
            r: inf
            e_eff: 0.5
            e_cap.max: 40000  # kW
        costs:
            monetary:
                e_cap: 750  # USD per kW
                om_fuel: 0.02  # USD per kWh
    csp:
        name: 'Concentrating solar power'
        color: '#99CB48'
        stack_weight: 100
        parent: supply_plus
        carrier_out: power
        constraints:
            use_s_time: true
            s_time.max: 24
            s_loss: 0.002
            r: file  # Will look for `csp_r.csv` in data directory
```

```
            e_eff: 0.4
            p_eff: 0.9
            r_area.max: inf
            e_cap.max: 10000
        costs:
            monetary:
                s_cap: 50
                r_area: 200
                r_cap: 200
                e_cap: 1000
                om_var: 0.002
        depreciation:
            monetary:
                interest: 0.12

    ##
    # Storage
    ##
    battery:
        name: 'Battery storage'
        color: '#DC5CE5'
        parent: storage
        carrier: power
        constraints:
            e_cap.max: 1000  # kW
            s_cap.max: inf
            c_rate: 4
            e_eff: 0.95  # 0.95 * 0.95 = 0.9025 round trip efficiency
            s_loss: 0  # No loss over time assumed
        costs:
            monetary:
                s_cap: 200  # USD per kWh storage capacity
    ##
    # Demand
    ##
    demand_power:
        name: 'Power demand'
        parent: demand
        carrier: power
    unmet_demand_power:
        name: 'Unmet power demand'
        parent: unmet_demand
        carrier: power

    ##
    # Transmission
    ##
    ac_transmission:
        name: 'AC power transmission'
        parent: transmission
        carrier: power
        constraints:
            e_eff: 0.85
        costs:
            monetary:
                e_cap: 200
                om_var: 0.002
```

locations.yaml:

```
##
# LOCATIONS
##

locations:
    region1:
        techs: ['demand_power', 'unmet_demand_power', 'ccgt']
        override:
            demand_power:
                x_map: 'region1: demand'
                constraints:
                    r: file=demand-1.csv
                    r_scale_to_peak: -40000
            ccgt:
                constraints:
                    e_cap.max: 30000 # increased to ensure no unmet_demand in first␣
↪timestep

    region2:
        techs: ['demand_power', 'unmet_demand_power', 'battery']
        override:
            demand_power:
                x_map: 'region2: demand'
                constraints:
                    r: file=demand-2.csv
                    r_scale_to_peak: -5000

    region1-1,region1-2,region1-3:
        within: region1
        techs: ['csp']

##
# TRANSMISSION CAPACITIES
##

links:
    region1,region2:
        ac_transmission:
            constraints:
                e_cap.max: 10000



##
# METADATA
##

metadata:
    # map boundary defined by the lower left and upper right of the square
    map_boundary:
        lower_left:
            lat: 35
            lon: -10
        upper_right:
            lat: 45
            lon: 5
```

```
    location_coordinates:  # lat, lon coordinates in a dictionary
        region1: {lat: 40, lon: -2}
        region2: {lat: 40, lon: -8}
        region1-1: {lat: 41, lon: -2}
        region1-2: {lat: 39, lon: -1}
        region1-3: {lat: 39, lon: -2}
```

Run settings

run.yaml:

```
##
# RUN SETTINGS
##

name: "Test run"  # Run name -- distinct from model name!

model: 'model_config/model.yaml'

output:  # Only used if run via the 'calliope run' command-line tool
    format: csv  # Choices: netcdf, csv
    path: 'Output'  # Will be created if it doesn't exist

mode: plan  # Choices: plan, operate

solver: glpk

##
# PARALLEL RUN SETTINGS
##

# Ignored unless run via the 'calliope generate' tool

parallel:
    name: example-model-national
    environment: bsub  # Choices: bsub, qsub
    pre_run:  # Commands to run before executing model
    post_run:  # Commands to run after executing model
    iterations:
        - subset_t: ['2005-01-01', '2005-01-31']
          override.locations.r1.techs: ['demand', 'unmet_demand', 'ccgt']
        - subset_t: ['2005-02-01', '2005-02-31']
          override.locations.r1.techs: ['demand', 'unmet_demand']
    resources:  # Request resources on a computing cluster
        threads:  # Non-default number of threads
        wall_time:  # Run time (minutes)
        memory:  # Working memory (MB)

##
# TIME RESOLUTION ADJUSTMENT
##

# time:
#     resolution: 6  # Reduce rest of data to 6-hourly timesteps
#     masks:  # Look for week where CSP output is minimal
#         - function: mask_extreme_week
#           options: {what: min, tech: csp}
```

```
#

##
# SUBSETS
##

# Leave any of these empty to disable subsetting

subset_y: []  # Subset of technologies
subset_x: []  # Subset of locations
subset_t: ['2005-01-01', '2005-01-05']  # Subset of timesteps

##
# MODEL SETTINGS OVERRIDE
##

# Override anything in the model configuration

override:

##
# DEBUG OPTIONS
##

debug:
    keep_temp_files: false  # Keep temporary files
    symbolic_solver_labels: false # Use human-readable component labels? (slower)
```

1.11.2 Urban-scale example

Available as `calliope.examples.UrbanScale`.

Model settings

`model.yaml`:

```
##
# IMPORT OTHER FILES
##

# Can either be paths relative to this file, or absolute paths

import:
    - 'techs.yaml'
    - 'locations.yaml'

##
# MODEL NAME
##

name: "Urban scale example model"


##
# DATASET PATH
```

```
##

# Can either be a path relative to this file, or an absolute path

data_path: 'data'

##
# OBJECTIVE FUNCTION
##

# 'constraints.objective.objective_cost_minimization' is used by default
# objective:

##
# ADDITIONAL CONSTRAINTS
##

constraints:
    - constraints.optional.max_r_area_per_loc

# OTHER MODEL-WIDE OPTIONS
##

system_margin:
    power: 0
    heat: 0
    cooling: 0
```

techs.yaml:

```
##
# TECHNOLOGY DEFINITIONS
##

techs:

##-GRID SUPPLY-##

    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 # 10p/kWh electricity price #ppt

    supply_gas:
        name: 'Natural gas import'
        parent: supply
        carrier: gas
        constraints:
            r: inf
            e_cap.max: 2000
        costs:
```

```
            monetary:
                e_cap: 1
                om_fuel: 0.025 # 2.5p/kWh gas price #ppt

##-Renewables-##

    pv:
        name: 'Solar photovoltaic power'
        color: '#99CB48'
        stack_weight: 100
        parent: supply
        export: true
        carrier_out: power
        constraints:
            r: file  # Will look for `pv_r.csv` in data directory - already accounted␣
↪for panel efficiency
            e_eff: 0.85
            e_cap.max: 250
            r_area.max: 1500
        costs:
            monetary:
                e_cap: 1350
# Conversion

    boiler:
        name: 'Natural gas boiler'
        stack_weight: 100
        parent: conversion
        carrier_out: heat
        carrier_in: gas
        constraints:
            e_cap.max: 600
            e_eff: 0.85

# Conversion_plus

    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: 1300
            e_eff: 0.405
        costs:
            monetary:
                e_cap: 750
                om_var: 0.004 # .4p/kWh for 4500 operating hours/year
                export: file=export_power.csv

##-DEMAND-##

    demand_power:
```

```
        name: 'Electrical demand'
        parent: demand
        carrier: power

    unmet_demand_power:
        name: 'Unmet electrical demand'
        parent: unmet_demand
        carrier: power

    demand_heat:
        name: 'Heat demand'
        parent: demand
        carrier: heat

    unmet_demand_heat:
        name: 'Unmet heat demand'
        parent: unmet_demand
        carrier: heat

##-DISTRIBUTION-##

    power_lines:
        name: 'Electrical power distribution'
        parent: transmission
        carrier: power
        constraints:
            e_cap.max: 2000
            e_eff: 0.98
        costs_per_distance:
            monetary:
                e_cap: 0.01

    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
```

locations.yaml:

```
##
# LOCATIONS
##

locations:
    X1:
        techs: ['chp', 'pv',
         'supply_grid_power', 'supply_gas',
         'demand_power', 'demand_heat',
         'unmet_demand_power', 'unmet_demand_heat']
        available_area: 500
        override:
```

```
            demand_power.constraints.r: file=demand_power.csv
            demand_heat.constraints.r: file=demand_heat.csv
            supply_grid_power.costs.monetary.e_cap: 100 # cost of transformers

    X2:
        techs: ['boiler', 'pv',
         'supply_gas',
         'demand_power', 'demand_heat',
         'unmet_demand_power', 'unmet_demand_heat'
         ]
        available_area: 1300
        override:
            demand_power.constraints.r: file=demand_power.csv
            demand_heat.constraints.r: file=demand_heat.csv
            boiler.costs.monetary.e_cap: 43.1 # different boiler costs
            pv.costs.monetary:
                        om_var: -0.0203 # revenue for just producing electricity
                        export: -0.0491 # FIT return for PV export

    X3:
        techs: ['boiler', 'pv',
         'supply_gas',
         'demand_power', 'demand_heat',
         'unmet_demand_power', 'unmet_demand_heat'
         ]
        available_area: 900
        override:
            demand_power.constraints.r: file=demand_power.csv
            demand_heat.constraints.r: file=demand_heat.csv
            boiler.costs.monetary.e_cap: 78 # different boiler costs
            pv:
                constraints:
                    e_cap.max: 50 # changing tariff structure below 50kW
                costs.monetary:
                        om_fixed: -80.5 # reimbursement per kWp from FIT


    N1: # location for branching heat transmission network
        techs: ['heat_pipes']

links:
    X1,X2:
        power_lines:
            distance: 10
    X1,X3:
        power_lines:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
    X1,N1:
        heat_pipes:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
    N1,X2:
        heat_pipes:
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting
    N1,X3:
        heat_pipes:
```

```
            constraints: # nothing to define, but model requires a key at this level␣
↪of nesting

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:
        X1: {x: 2, y: 7}
        X2: {x: 8, y: 7}
        X3: {x: 5, y: 3}
        N1: {x: 5, y: 7}
```

Run settings

run.yaml:

```
##
# RUN SETTINGS
##

name: "Test run"  # Run name -- distinct from model name!

model: 'model_config/model.yaml'

output:  # Only used if run via the 'calliope run' command-line tool
    format: csv  # Choices: netcdf, csv
    path: 'Output'  # Will be created if it doesn't exist

mode: plan  # Choices: plan, operate

solver: glpk

##
# TIME RESOLUTION ADJUSTMENT
##

# time:
#     resolution: 6  # Reduce rest of data to 6-hourly timesteps
#     masks:  # Look for week where CSP output is minimal
#         - function: mask_extreme_week
#           options: {what: min, tech: csp}
#

##
# SUBSETS
##

# Leave any of these empty to disable subsetting

subset_y: []  # Subset of technologies
subset_x: []  # Subset of locations
```

```
subset_t: ['2005-01-01', '2005-01-02']  # Subset of timesteps

##
# MODEL SETTINGS OVERRIDE
##

# Override anything in the model configuration

override:

##
# DEBUG OPTIONS
##

debug:
    keep_temp_files: false  # Keep temporary files
    symbolic_solver_labels: false # Use human-readable component labels? (slower)
```

1.12 Development guide

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.

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.

1.12.1 Installing a development version

As when installing a stable version, using `conda` is recommended.

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.yml -n calliope_dev
$ source activate calliope_dev
```

Then install Calliope itself with pip:

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

1.12.2 Creating modular extensions

Constraint generator functions

By making use of the ability to load custom constraint generator functions (see *Loading optional constraints*), a Calliope model can be extended by additional constraints easily without modifying the core code.

Constraint generator functions are called during construction of the model with the *Model* object passed as the only parameter.

The Model object provides, amongst other things:

- The Pyomo model instance, under the property m
- The model data under the data property
- An easy way to access model configuration with the *get_option()* method

A constraint generator function can add constraints, parameters, and variables directly to the Pyomo model instance (Model.m). Refer to the Pyomo documentation for information on how to construct these model components.

The default cost-minimizing objective function provides a good example:

```
import pyomo.core as po  # pylint: disable=import-error

def objective_cost_minimization(model):
    """
    Minimizes total system monetary cost.
    Used as a default if a model does not specify another objective.

    """
    m = model.m

    def obj_rule(m):
        return sum(model.get_option(y + '.weight') *
                   sum(m.cost[y, x, 'monetary']
                    for x in m.x) for y in m.y)

    m.obj = po.Objective(sense=po.minimize, rule=obj_rule)
    m.obj.domain = po.Reals
```

See the source code of the *ramping_rate()* function for a more elaborate example.

The process of including custom, optional constraints is as follows:

First, create the source code (see e.g. the above example for the ramping_rate function) in a file, for example my_constraints.py

Then, assuming your custom constraint generator function is called my_first_custom_constraint and is defined in my_constraints.py, you can tell Calliope to load it by adding it to the list of optional constraints in your model configuration as follows:

```
constraints:
    - constraints.optional.ramping_rate
    - my_constraints.my_first_custom_constraint
```

This assumes that the file my_constraints.py is importable when the model is run. It must therefore either be in the directory from which the model is run, installed as a Python module (see this document on how to create importable and installable Python packages), or the Python import path has to be adjusted according to the official Python documentation.

Subsets

Calliope internally builds many subsets to better manage constraints, in particular, subsets of different groups of technologies. These subsets can be used in the definition of constraints and are used extensively in the definition of Calliope's built-in constraints. See the detailed definitions in calliope.sets, an overview of which is included here.

Main sets & sub-sets

Technologies:

- **m.y_demand: all demand sources**
  - m.y_sd_r_area: if any r_area constraints are defined (shared)
  - m.y_sd_finite_r: if finite resource limit is defined (shared)
- **m.y_supply: all basic supply technologies**
  - m.y_sd_r_area: if any r_area constraints are defined (shared)
  - m.y_sd_finite_r: if finite resource limit is defined (shared)
- m.y_storage: specifically storage technologies
- **m.y_supply_plus: all supply+ technologies**
  - m.y_sp_r_area: If any r_area constraints are defined
  - m.y_sp_finite_r: if finite resource limit is defined
  - m.y_sp_r2: if secondary resource is allowed
- m.y_conversion: all basic conversion technologies
- **m.y_conversion_plus: all conversion+ technologies**
  - m.y_cp_2out: secondary carrier(s) out
  - m.y_cp_3out: tertiary carrier(s) out
  - m.y_cp_2in: secondary carrier(s) in
  - m.y_cp_3in: tertiary carrier(s) in
- m.y_transmission: all transmission technologies
- m.y_unmet: dummy supply technologies to log

Locations:

- m.x_trans: all transmission locations
- m.x_r: all locations which act as system sources/sinks
- m.x_store: all locations in which storage is allowed

Shared subsets

- **m.y_finite_r: shared between y_demand, y_supply, and y_supply_plus. Contains:**
  - m.y_sd_finite_r
  - m.y_sp_finite_r
- **m.y_r_area: shared between y_demand, y_supply, and y_supply_plus. Contains:**
  - m.y_sd_r_area
  - m.y_sp_r_area

Meta-sets

Technologies:

- **`m.y`: all technologies, includes:**
  - `m.y_demand`
  - `m.y_supply`
  - `m.y_storage`
  - `m.y_supply_plus`
  - `m.y_conversion`
  - `m.y_conversion_plus`
  - `m.y_transmission`
- **`m.y_sd`: all basic supply & demand technologies, includes:**
  - `m.y_demand`
  - `m.y_supply`
- **`m.y_store`: all technologies that have storage capabilities, includes:**
  - `m.y_storage`
  - `m.y_supply_plus`

Locations:

- **`m.x`: all locations, includes:**
  - `m.x_trans`
  - `m.x_r`
  - `m.x_store`

Time functions and masks

Like custom constraint generator functions, custom functions that adjust time resolution can be loaded dynamically during model initialization. By default, Calliope first checks whether the name of a function or time mask refers to a function from the *`calliope.time_masks`* or `calliope.time_functions` 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: {...}
```

1.12.3 Profiling

To profile a Calliope run with the built-in national-scale example model, then visualize 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 visualizing:

- Interactive visualization 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
```

1.12.4 Checklist for new release

Pre-release

- Make sure all unit tests pass
- 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

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_run=None*, *override=None*)

Calliope model.

Parameters **config_run** : str or AttrDict, optional

Path to YAML file with run settings, or AttrDict containing run settings. If not given, the included default run and model settings are used.

override : AttrDict, optional

Provide any additional options or override options from `config_run` by passing an AttrDict of the form `{'model_settings': 'foo.yaml'}`. Any option possible in `run.yaml` can be specified in the dict, inluding `override.` options.

check_and_set_export()

In instances where a technology is allowing export, e.g. techs.ccgt.export: true then change 'true' to the carrier of that technology.

functionality_switch(*func_name*)

Check if a given functionality of the model is required, based on whether there is any reference to it in model configuration that isn't defaults.

Args:

- func_name: str; the funcitonality to check

Returns: bool; Whether the functionality is switched is on (True) or off (False)

generate_model(*t_start=None*)
Generate the model and store it under the property *m*.

Args: t_start : if self.mode == 'operate', this must be specified, but that is done automatically via solve_iterative() when calling run()

get_capacity_factor()
Get capacity factor.

NB: Only production, not consumption, is used in calculations.

get_carrier(*y*, *direction*, *level=None*, *primary=False*, *all_carriers=False*)
Get the `carrier_in` or `carrier_out` of a technology in the model

Parameters y: str

technology

direction: str, 'in' or 'out'

For *carrier_in* and *carrier_out* repectively

level: int; 2 or 3; optional, default = None

for conversion_plus technologies, define the carrier level if not top level, e.g. level=3 gives *carrier_out_3*

primary: bool, optional, default = False

give *primary carrier* for a given technology, which is a carrier in *carrier_out* given ass *primary carrier* in the technology definition

all_carriers: bool, optional, default = False

give all carriers for tech y and given direction. For conversion_plus technologies, this will give an array of carriers, if more than one carrier has been defined in the given direction. All levels are combined.

get_cp_carriers(*y*, *x=None*, *direction='out'*)
Find all carriers for conversion_plus technology & return the primary output carrier as string and all other output carriers as list of strings

get_distances()
Where distances are not given for links, use any metadata to fill in the gap. Distance calculated using vincenty inverse formula (given in utils module).

get_eff_ref(*var*, *y*, *x=None*)
Get reference efficiency, falling back to efficiency if no reference efficiency has been set.

get_group_members(*group*, *in_model=True*, *head_nodes_only=True*, *expand_transmission=True*)
Return the member technologies of a group. If `in_model` is True, only technologies (head nodes) in use in the current model are returned.

Returns:

- A list of group members if there are any.
- If a group has no members (is only member of other groups, i.e. a head node), a list with a single item containing only the group/technology itself.
- An empty list if the group is defined but not allowed in the current model.
- None if the group doesn't exist.

Other arguments:

head_nodes_only [if True, don't return intermediate] groups, i.e. technology definitions that are inherited from. Setting this to False only makes sense if in_model is also False, because in_model=True implies that only head nodes are returned.

expand_transmission [if True, return in-model] transmission technologies in the form `tech:location`.

get_levelized_cost()
Get levelized costs.

NB: Only production, not consumption, is used in calculations.

get_option(*option*, *x=None*, *default=None*, *ignore_inheritance=False*)
Retrieves options from model settings for the given tech, falling back to the default if the option is not defined for the tech.

If `x` is given, will attempt to use location-specific override from the location matrix first before falling back to model-wide settings.

If `default` is given, it is used as a fallback if no default value can be found in the regular inheritance chain. If `default` is None and the regular inheritance chain defines no default, an error is raised.

If `ignore_inheritance` is True, the default is immediately used instead of a search through the inheritance chain if the option has not been set for the given tech.

If the first segment of the option contains ':', it will be interpreted as implicit tech subsetting: e.g. asking for 'hvac:r1' implicitly uses 'hvac:r1' with the parent 'hvac', even if that has not been defined, to search the option inheritance chain.

Examples:

- `model.get_option('ccgt.costs.om_var')`
- `model.get_option('csp.weight')`
- `model.get_option('csp.r', x='33')`
- `model.get_option('ccgt.costs.om_var', default='defaults.costs.om_var')`

get_parent(*y*)
Returns the abstract base technology from which `y` descends.

get_t(*timestamp*, *offset=0*)
Get a timestamp before/after (by offset) from the given timestamp in the model's set of timestamps. Raises ModelError if out of bounds.

get_timeres(*verify=False*)
Returns resolution of data in hours.

If `verify=True`, verifies that the entire file is at the same resolution. `self.get_timeres(verify=True)` can be called after Model initialization to verify this.

get_totals(*t_subset=None*, *apply_weights=True*)
Get total produced and consumed per technology and location.

get_var(*var*, *dims=None*, *standardize_coords=True*)
Return output for variable *var* as a pandas.Series (1d), pandas.Dataframe (2d), or xarray.DataArray (3d and higher).

Parameters **var** : variable name as string, e.g. 'es_prod'

dims : list, optional

indices as strings, e.g. ('y', 'x', 't'); if not given, they are auto-detected

initialize_timeseries()
Find any constraints/costs values requested as from 'file' in YAMLs and store that information.

ischild(*y*, *of*)
Returns True if `y` is a child of `of`, else False

load_results()
Load results into model instance for access via model variables.

prev_t(*timestamp*)
Return the timestep prior to the given timestep.

process_solution()
Called from both load_solution() and load_solution_iterative()

read_data()
Populate parameter data from CSV files or model configuration.

run(*iterative_warmstart=True*)
Instantiate and solve the model

save_solution(*how*)
Save model solution. `how` can be 'netcdf' or 'csv'

scale_to_peak(*df*, *peak*, *scale_time_res=True*)
Returns the given dataframe scaled to the given peak value.

If `scale_time_res` is True, the peak is multiplied by the model's time resolution. Set it to False to scale things like efficiencies.

set_option(*option*, *value*, *x=None*)
Set `option` to `value`. Returns None on success.

A default can be set by passing an option like `defaults.constraints.e_eff`.

solve(*warmstart=False*)

Args:

warmstart [(default False) re-solve an updated model] instance

Returns: None

solve_iterative(*iterative_warmstart=True*)
Solve iterative by updating model parameters.

By default, on optimizations subsequent to the first one, warmstart is used to speed up the model generation process.

Returns None on success, storing results under self.solution

2.1.2 Constraints

`calliope.constraints.objective.`**objective_cost_minimization**(*model*)
Minimizes total system monetary cost. Used as a default if a model does not specify another objective.

`calliope.constraints.base.`**generate_variables**(*model*)
Defines variables:

- r: resource -> tech (+ production
- r_area: resource collector area
- r2: secondary resource -> storage (+ production)

- c_prod: tech -> carrier (+ production)
- c_con: tech <- carrier (- consumption)
- s_cap: installed storage capacity
- r_cap: installed resource <-> storage conversion capacity
- e_cap: installed storage <-> grid conversion capacity (gross)
- r2_cap: installed secondary resource conversion capacity
- cost: total costs
- cost_con: construction costs
- cost_op_fixed: fixed operation costs
- cost_op_var: variable operation costs
- cost_op_fuel: primary resource fuel costs
- cost_op_r2: secondary resource fuel costs

calliope.constraints.base.**get_constraint_param**(*model*, *param_string*, *y*, *x*, *t*)
Function to get values for constraints which can optionally be loaded from file (so may have time dependency).

model = calliope model param_string = constraint as string y = technology x = location t = timestep

calliope.constraints.base.**get_cost_param**(*model*, *param_string*, *k*, *y*, *x*, *t*)
Function to get values for constraints which can optionally be loaded from file (so may have time dependency).

model = calliope model cost = cost name, e.g. 'om_fuel' k = cost type, e.g. 'monetary' y = technology x = location t = timestep

calliope.constraints.base.**node_constraints_transmission**(*model*)
Constrain e_cap symmetrically for transmission nodes. Transmission techs only.

calliope.constraints.planning.**node_constraints_build_total**(*model*)

calliope.constraints.planning.**system_margin**(*model*)

calliope.constraints.optional.**group_fraction**(*model*)
Constrain groups of technologies to reach given fractions of e_prod.

calliope.constraints.optional.**max_r_area_per_loc**(*model*)
`r_area` of all technologies requiring physical space cannot exceed the available area of a location. Available area defined for parent locations (in which there are locations defined as being 'within' it) will set the available area limit for the sum of all the family (parent + all desecendants).

To define, assign a value to `available_area` for a given location, e.g.:

```
locations:
    r1:
        techs: ['csp']
        available_area: 100000
```

To avoid including descendants in area limitation, `ignore_descendants` can be specified for the location, in the same way as `available_area`.

calliope.constraints.optional.**ramping_rate**(*model*)
Ramping rate constraints.

Depends on: node_energy_balance, node_constraints_build

2.1.3 Time series

calliope.time_funcs.**apply_clustering**(*data*, *timesteps*, *clustering_func*, *how*, *normalize=True*, ***kwargs*)

Apply the given clustering function to the given data.

Parameters **data** : xarray.Dataset

timesteps : pandas.DatetimeIndex or list of timesteps or None

clustering_func : str

Name of clustering function.

how : str

How to map clusters to data. 'mean' or 'closest'.

normalize : bool, optional

If True (default), data is normalized before clustering is applied, using *normalized_copy()*.

****kwargs** : optional

Arguments passed to clustering_func.

Returns **data_new_scaled** : xarray.Dataset

calliope.time_funcs.**drop**(*data*, *timesteps*, *padding=None*)

Drop timesteps from data, with optional padding around into the contiguous areas encompassed by the timesteps.

calliope.time_funcs.**normalized_copy**(*data*)

Return a copy of data, with the absolute taken and normalized to 0-1.

The maximum across all regions and timesteps is used to normalize.

calliope.time_masks.**extreme**(*data*, *tech*, *var='r'*, *how='max'*, *length='1D'*, *n=1*, *groupby_length=None*, *locations=None*, *padding=None*)

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

Parameters **data** : xarray.Dataset

tech : str

Technology whose *var* to find extreme for.

var : str, optional

default 'r'

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.

locations : list, optional

List of locations to use, if None, uses all available locations.

padding : int, optional

Pad beginning and end of the unmasked area by the number of timesteps given.

normalize : bool, optional

If True (default), data is normalized using *normalized_copy()*.

calliope.time_masks.**zero**(*data*, *tech*, *var='r'*, *locations=None*)
Returns timesteps where `var` for the technology `tech` across the given list of `locations` is zero.

If `locations` not given, uses all available locations.

calliope.time_clustering.**cophenetic_corr**(*X*, *Z*)
Get the Cophenetic Correlation Coefficient of a clustering with help of the cophenet() function. This (very very briefly) compares (correlates) the actual pairwise distances of all your samples to those implied by the hierarchical clustering. The closer the value is to 1, the better the clustering preserves the original distances.

Source: https://joernhees.de/blog/2015/08/26/scipy-hierarchical-clustering-and-dendrogram-tutorial/

calliope.time_clustering.**fancy_dendrogram**(**args*, ***kwargs*)
Code adapted from: https://joernhees.de/blog/2015/08/26/scipy-hierarchical-clustering-and-dendrogram-tutorial/

calliope.time_clustering.**get_clusters_hierarchical**(*data*, *tech=None*, *max_d=None*, *k=None*)

Parameters **data** : xarray.Dataset

Should be normalized

max_d : float or int, optional

Max distance for returning clusters.

k : int, optional

Number of desired clusters.

Returns clusters

X

Z

calliope.time_clustering.**get_clusters_kmeans**(*data*, *tech=None*, *timesteps=None*, *k=5*)

Parameters **data** : xarray.Dataset

Should be normalized

Returns **clusters** : dataframe

Indexed by timesteps and with locations as columns, giving cluster membership for first timestep of each day.

centroids

calliope.time_clustering.**map_clusters_to_data**(*data*, *clusters*, *how*)
Returns a copy of data that has been clustered.

Parameters **how** : str

How to select data from clusters. Can be mean (centroid) or closest.

2.1.4 Reading results

calliope.read.**read_dir**(*directory*)
Combines output files from *directory* and return an AttrDict containing them all.

If a solution is missing or there is an error reading it, an empty AttrDict is added to the results in its stead and the error is logged.

calliope.read.**read_netcdf**(*path*)
Read model solution from NetCDF4 file

2.1.5 Analyzing results

calliope.analysis.**areas_below_resolution**(*solution*, *resolution*)
Returns a list of (start, end) timestamp tuples delimiting those areas in the solution below the given timestep resolution (in hours).

calliope.analysis.**get_delivered_cost**(*solution*, *cost_class='monetary'*, *carrier='power'*, *count_unmet_demand=False*)
Get the levelized cost per unit of energy delivered for the given `cost_class` and `carrier`.

Parameters **solution** : solution container

cost_class : str, default 'monetary'

carrier : str, default 'power'

count_unmet_demand : bool, default False

Whether to count the cost of unmet demand in the final delivered cost.

calliope.analysis.**get_domestic_supply_index**(*solution*)
Assuming that `solution` specifies a `domestic` cost class to give each technology a domesticity score, return the total domestic supply index for the given solution.

calliope.analysis.**get_group_share**(*solution*, *techs*, *group*, *var='e_prod'*)
From `solution.summary`, get the share of the given list of `techs` from the total for the given `group`, for the given `var`.

calliope.analysis.**get_hhi**(*solution, shares_var='e_cap', exclude_patterns=['unmet_demand']*)
Returns the Herfindahl-Hirschmann diversity index.

$HHI = \sum_{i=1}^{I} p_i^2$

where p_i is the percentage share of each technology i (0-100).

HHI ranges between 0 and 10,000. A value above 1800 is considered a sign of a concentrated market.

calliope.analysis.**get_levelized_cost**(*solution*, *cost_class='monetary'*, *carrier='power'*, *groups=None*, *locations=None*, *unit_multiplier=1.0*)
Get the levelized cost per unit of energy produced for the given `cost_class` and `carrier`, optionally for a subset of technologies given by `groups` and a subset of `locations`.

Parameters **solution** : solution container

cost_class : str, default 'monetary'

carrier : str, default 'power'

groups : list, default None

Limit the computation to members of the given groups (see the groups table in the solution for valid groups). Defaults to ['supply', 'supply_plus'] if not given.

locations : str or iterable, default None

Limit the computation to the given location or locations.

unit_multiplier : float or int, default 1.0

Adjust unit of the returned cost value. For example, if model units are kW and kWh, `unit_multiplier=1.0` will return cost per kWh, and `unit_multiplier=0.001` will return cost per MWh.

`calliope.analysis.`**`get_swi`**(*solution, shares_var='e_cap', exclude_patterns=['unmet_demand']*)

Returns the Shannon-Wiener diversity index.

$SWI = -1 \times \sum_{i=1}^{I} p_i \times \ln(p_i)$

where where I is the number of categories and p_i is each category's share of the total (between 0 and 1).

SWI is zero when there is perfect concentration.

`calliope.analysis.`**`get_unmet_demand_hours`**(*solution*, *carrier='power'*, *details=False*)

Get information about unmet demand from `solution`.

Parameters **solution** : solution container

carrier : str, default 'power'

details : bool, default False

By default, only the number of hours with unmet are returned. If details is True, a dict with 'hours', 'timesteps', and 'dates' keys is returned instead.

`calliope.analysis.`**`map_results`**(*results*, *func*, *as_frame=False*)

Applies `func` to each model solution in `results`, returning a pandas DataFrame (if as_frame is True) or Series, indexed by the run names (if available).

`calliope.analysis.`**`plot_carrier_production`**(*solution*, *carrier='power'*, *subset={}*, ***kwargs*)

Generate a stackplot of the production by the given `carrier`.

Parameters **solution** : model solution xarray.Dataset

carrier : str, optional

Name of the carrier to plot, default 'power'.

subset : dict, optional

Specify an additional subset of Dataset coordinates, for example, dict(t=slice('2005-02-01', '2005-02-10').

****kwargs** : optional

Passed to `plot_timeseries`.

`calliope.analysis.`**`plot_installed_capacities`**(*solution, tech_types=['supply', 'supply_plus', 'conversion', 'conversion_plus', 'storage'], unit_multiplier=1.0, unit_label='kW', **kwargs*)

Plot installed capacities (`e_cap`) with a bar plot.

Parameters **solution** : model solution xarray.Dataset

tech_types : list, optional

Technology types to include in the plot. Default is ['supply', 'supply_plus', 'conversion', 'conversion_plus', 'storage']

unit_multiplier : float or int, optional

Multiply installed capacities by this value for plotting. Defaults to 1.0

unit_label : str, optional

Label for capacity values. Default is 'kW', adjust this when changing `unit_multiplier`.

****kwargs** : optional

are passed to `pandas.DataFrame.plot()`

`calliope.analysis.`**`plot_timeseries`**(*solution, data, carrier='power', demand='demand_power', tech_types=['supply', 'supply_plus', 'conversion', 'conversion_plus', 'storage', 'unmet_demand'], colormap=None, ticks=None, resample_options=None, resample_func=None, add_legend=True, ax=None*)

Generate a stackplot of `data` for the given `carrier`, plotting `demand` on top.

Use `plot_carrier_production` for a simpler way to plot production by a given carrier.

Parameters **solution** : model solution xarray.Dataset

data : xarray.Dataset

Subset of solution to plot.

carrier : str, optional

Name of the carrier to plot, default 'power'.

demand : str, optional

Name of a demand tech whose time series to plot on top, default 'demand_power'.

tech_types : list, optional

Technology types to include in the plot. Default list is ['supply', 'supply_plus' , 'conversion', 'storage', 'unmet_demand'].

colormap : matplotlib colormap, optional

Colormap to use. If not given, the colors specified for each technology in the solution's metadata are used.

ticks : str, optional

Where to draw x-axis (time axis) ticks. By default (None), auto-detects, but can manually set to either 'hourly', 'daily', or 'monthly'.

resample_options : dict, optional

Give options for pandas.DataFrame.resample in a dict, to resample the entire time series prior to plotting. Both resample_options and resample_func must be given for resampling to happen. Default None.

resample_func : string, optional

Give the name of the aggregating function to use when resampling, e.g. "mean" or "sum". Default None.

`calliope.analysis.`**`plot_transmission`**(*solution, tech='ac_transmission', carrier='power', labels='utilization', figsize=(15, 15), fontsize=9, show_scale=True, ax=None, **kwargs*)

Plot transmission links on a map. Requires that model metadata have been defined with a lat/lon for each model location and a boundary for the map display.

Requires Basemap and NetworkX to be installed.

Parameters **solution** : solution container

tech : str, default 'ac_transmission'

Which transmission technology to plot.

carrier : str, default 'power'

Which carrier to plot transmission for.

labels : str, default 'utilization'

Determines how transmission links are labeled, either *transmission* or *utilization*.

figsize : (int, int), default (15, 15)

Size of resulting figure.

fontsize : int, default 9

Font size of figure labels.

show_scale : bool, default True

Plot a distance scale on the map.

ax : matplotlib axes, default None

****kwargs** : are passed to `analysis_utils.plot_graph_on_map()`

2.1.6 Utility classes: AttrDict, Parallelizer, Exceptions

class `calliope.utils.`**`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.

`as_dict`(*flat=False*)

Return the AttrDict as a pure dict (with nested dicts if necessary).

`copy`()

Override copy method so that it returns an AttrDict

`del_key`(*key*)

Delete the given key. Properly deals with nested keys.

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.

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.

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
```

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']}]`.

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
```

to_yaml(*path=None*, *convert_objects=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.

union(*other*, *allow_override=False*, *allow_replacement=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.

class `calliope.`**Parallelizer**(*target_dir*, *config_run=None*)
Arguments:

- `target_dir`: path to output directory for parallel runs.
- `config_run`: path to YAML file with run settings. If not given, the included example run.yaml is used.

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.`**ModelWarning**
ModelWarnings should be raised for possible model errors, but where execution can still continue.

2.2 Index

CHAPTER 3

Release history

3.1 Release History

3.1.1 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.

3.1.2 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

3.1.3 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

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

3.1.4 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

3.1.5 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

3.1.6 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

3.1.7 0.3.6 (2015-09-23)

- fixed Version 0.3.5 changes were not reflected in tutorial

3.1.8 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 *Technology 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

3.1.9 0.3.4 (2015-04-27)

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

3.1.10 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

3.1.11 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

3.1.12 0.3.1 (2015-01-06)

- Fixes to time_functions
- Other minor improvements and fixes

3.1.13 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

3.1.14 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

3.1.15 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

I

K

L

M

N

O

P

R

S

T

U

Z