Macro

Macro Energy Team

January 30, 2025

Contents

Part I

Welcome to Macro

Chapter 1

Introduction

Welcome to the MACRO documentation!

What is MACRO?

MAcro-Energy System **C**apacity Expansion & **R**esource **O**ptimization Model (MACRO) is a bottom-up, electricity-centric, macro-energy systems optimization model. It is designed to capture capacity investments, operations, and energy flows across and between multiple energy sectors and can be used to explore the impacts of different energy policies, technology costs, and other exogenous factors on the energy system.

The main features of MACRO include:

- Tailored **Benders decomposition** framework for optimization.
- **Graph-based representation** of the energy system, including nodes, storage units, edges/transmission lines, transformation nodes/conversion units.
- **"Plug and play" flexibility** for integrating various technologies and sectors (e.g., electricity, hydrogen, heat, and transport).
- Technologically rich, **granular temporal resolution** for detailed analysis.
- **Open-source** built using Julia and JuMP.

MACRO development strategy

MACRO has been designed and developed with three layers of abstractions in mind, each serving a different kind of user:

The following sections of the documentation are designed to serve the different needs of the different users:

- User Guide
- Modeler Guide
- Developer Guide

Index

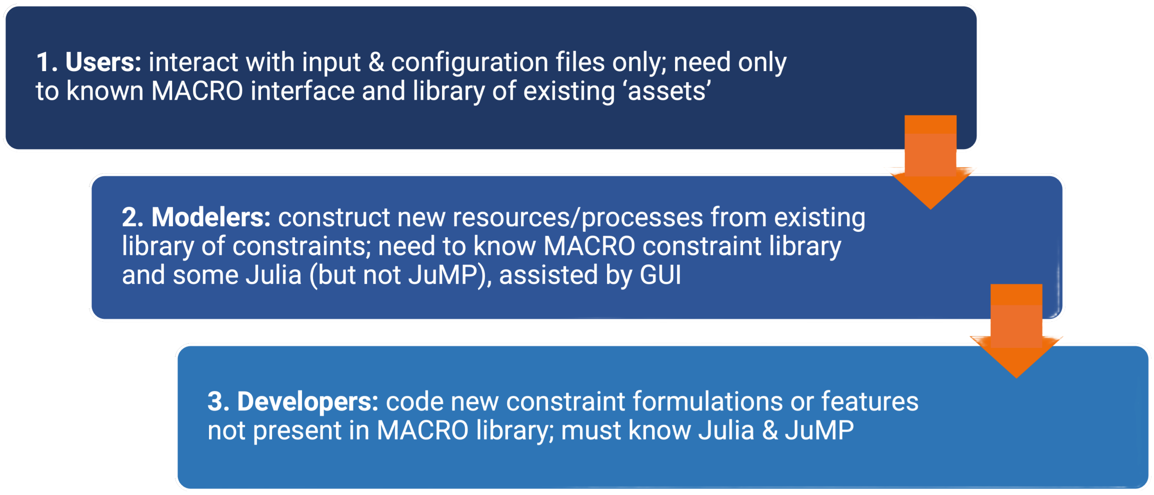
1. Users: interact with input & configuration files only; need only to known MACRO interface and library of existing 'assets'
2. Modelers: construct new resources/processes from existing library of constraints; need to know MACRO constraint library and some Julia (but not JuMP), assisted by GUI
3. Developers: code new constraint formulations or features not present in MACRO library; must know Julia & JuMP

Figure 1.1: MACRO architecture

Part II

Getting Started

Chapter 2

Overview

2.1 Overview

Multi-commodity flow network

MACRO is designed to represent energy systems in a detailed manner, capturing interactions among various sectors and technologies. The system is structured as a **multi-commodity flow network**, with each commodity having independent spatial and temporal scale:

As an example, the figure below illustrates a multi-plex network representing an energy system with electricity, natural gas, and CO2 sectors, with two natural gas power plants, and a solar panel. Blue nodes represent the electricity sector, red nodes represent natural gas, and yellow nodes represent CO2. The edges depict commodity flow, and squares represent transformation points.

As illustrated in the figures above, the core components of the model are:

1. **Vertices**: Represent **balance equations** and can correspond to transformations (linking two or more commodity networks), storage systems, or demand nodes (outflows):

   – **Transformations**:

     ∗ Special vertices that **convert** one commodity type into another, acting as bridges between sectors.

     ∗ They represent conversion processes defined by a set of **stoichiometric equations** specifying transformation ratios.

   – **Storage**:

     ∗ Stores commodities for future use.

     ∗ The flow of commodities into and out of storage systems is regulated by **Storage balance** equations.

   – **Nodes**:

     ∗ Represent geographical locations or zones, each associated with a commodity type.

     ∗ They can be of two types: demand nodes (outflows) or sources (inflows).

     ∗ **Demand balance** equations are used to balance the flow of commodities into and out of the node.

     ∗ They form the network for a specific sector (e.g., electricity network, hydrogen network, etc.).

2. **Edges**:

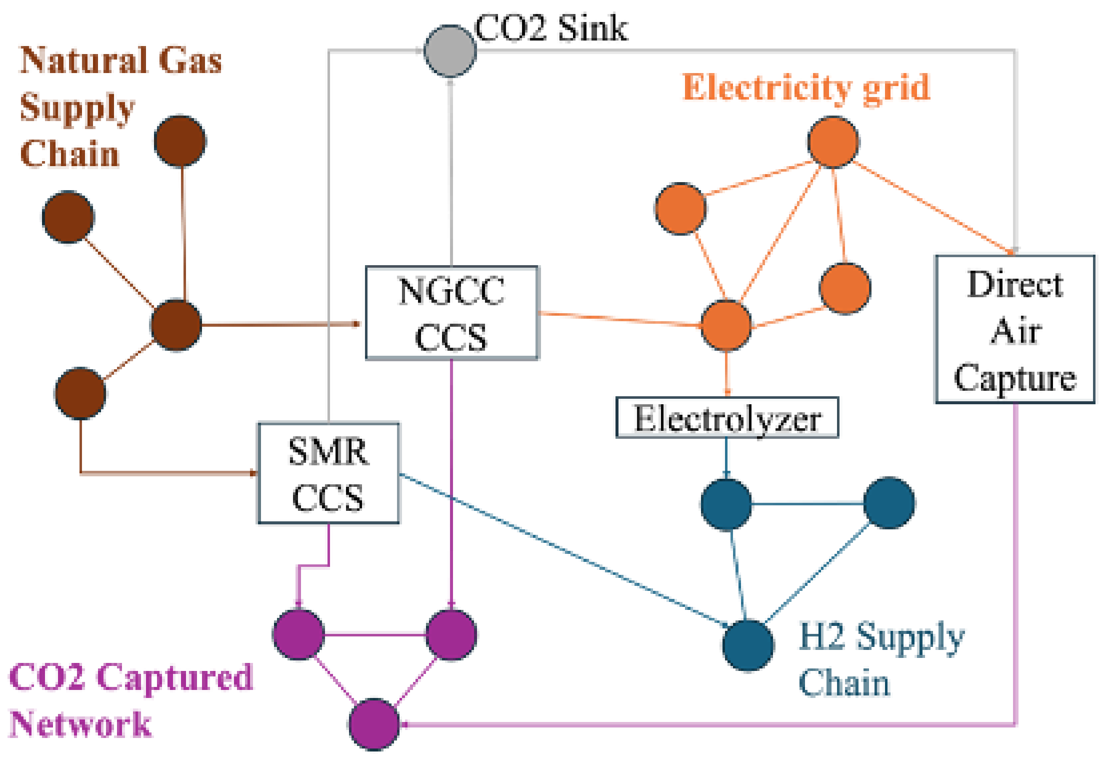

Figure 2.1: multi-commodity flow network

  - Depict the **flow** of a commodity into or out of a vertex.
  - Capacity sizing decisions, capex/opex, planning and operational constraints are associated with the edges.

3. **Assets**: Defined as a collection of edges and vertices. See MACRO Asset Library for a list of all the assets available in MACRO.

MACRO includes a library of assets (see MACRO Asset Library) and constraints (see MACRO Constraint Library), enabling **fast** and **flexible assembly** of new technologies and sectors.

Benders decomposition

MACRO is natively designed to create optimization models ready to be solved using the Benders decomposition framework and high-performance decomposition algorithms. This allows to solve large-scale problems with several sectors interacting together in a reasonable time frame.

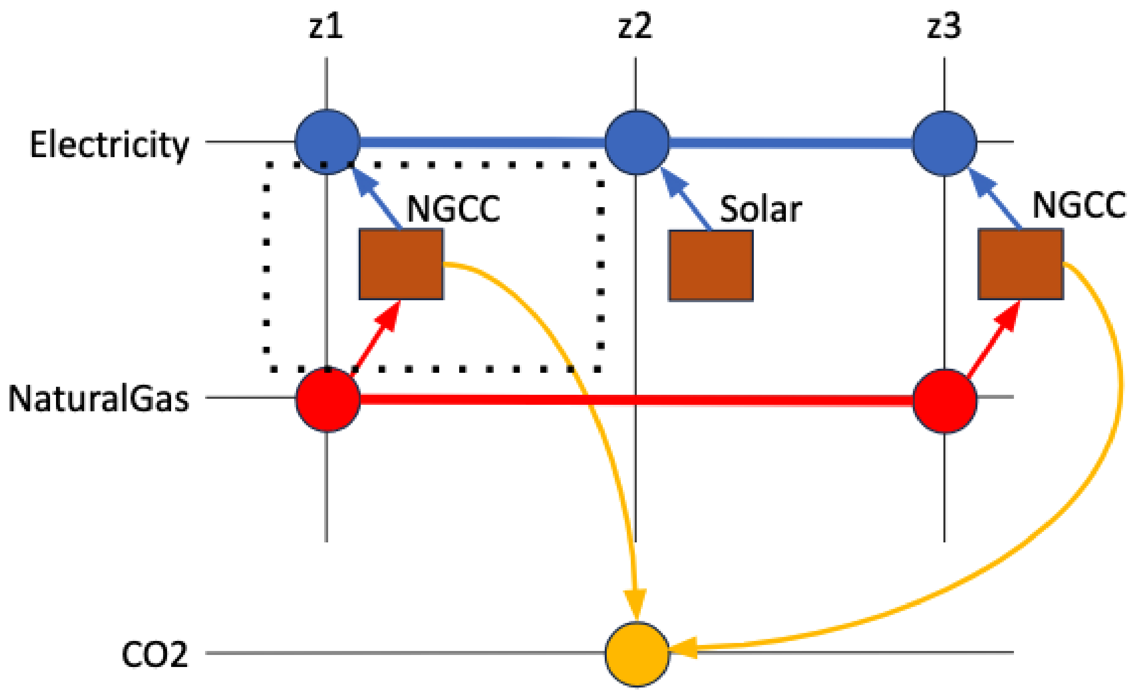

Figure 2.2: network

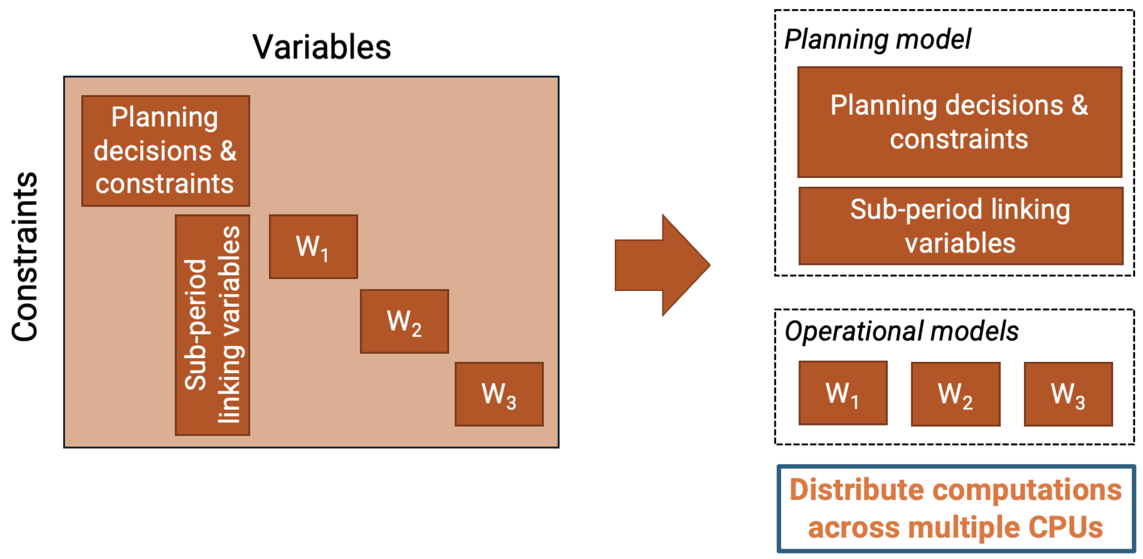
Variables
Constraints
Planning decisions & constraints
Sub-period linking variables
W_1
W_2
W_3
Planning model
Planning decisions & constraints
Sub-period linking variables
Operational models
W_1
W_2
W_3
Distribute computations across multiple CPUs

Figure 2.3: model_structure

Chapter 3

Installation

3.1 Installation

Requirements

- **Julia** 1.9 or later
- **Git** (to clone the repository)

Access to the MACRO repository

The following steps assume that the user has a GitHub account and a PAT (Personal Access Token) or SSH key pair. For more information on how to create a PAT or SSH key pair, please refer to the GitHub documentation.

Current version: 0.1.0: We also assume that the user has been added to the MACRO repository as a collaborator (the repository is private).

Installation steps

To install the MACRO package in Julia, we recommend following these steps:

- **Clone the MACRO repository**:

If you are using a PAT, you can use the following command:

```
git clone https://<your_username>:<your_pat>@github.com/macroenergy/Macro.git
```

If you are using an SSH key pair instead of a PAT, you can use the following command:

```
git clone git@github.com:macroenergy/Macro.git
```

Cloning a specific branch

If you want to clone a specific branch, you can use the -b flag:

```
git clone -b <branch_name>
↪  https://<your_username>:<your_pat>@github.com/macroenergy/Macro.git
```

- **Navigate to the cloned repository**:

```
cd Macro
```

- **Install MACRO and all its dependencies**:

```
julia --project -e 'using Pkg; Pkg.instantiate(); Pkg.precompile()'
```

- **Test the installation**:

Start Julia with the project environment in a terminal:

```
$ julia --project
```

Load the MACRO package in the Julia REPL:

```
using Macro
```

Editing the installation

If the user wants to edit the installation, for example, to install a specific version of a dependency, they can do so by following the steps below:

- Run a Julia session with the MACRO project environment activated:

```
$ cd Macro
$ julia --project
```

Alternatively, you can first run Julia and then enter the Pkg mode to activate the project environment:

```
$ cd Macro
$ julia
```

In the Julia REPL, enter the Pkg mode by pressing], then activate the project environment:

```
] activate .
```

- Use the Pkg mode to install or update a dependency:

```
] rm <dependency_name>
] add <dependency_name>@<version>
```

For instance, to install the JuMP package version v1.22.2, you can use the following commands:

```
] rm JuMP
] add JuMP@v1.22.2
```

Activating the project environment

When working with the MACRO package, always remember to activate the project environment before running any commands. This ensures that the correct dependencies are used and that the project is in the correct state.

To activate the project environment, you can use the following commands:

```
cd Macro
julia --project
```

or

```
cd Macro
julia
] activate .
```

Part III

Tutorials

Chapter 4

Getting Started

4.1 Tutorial 0: Getting Started with MACRO

Interactive Notebook

The interactive version of this tutorial can be found here.

This tutorial will guide you through the steps to install MACRO, a solver, and all the necessary dependencies.

Installation

Before installing MACRO, make sure you have the following requirements installed:

Requirements

- **Julia**: you can download it here.
- **Git**: you can download it here.
- (optional) **Jupyter Notebook**: you can install it using the following command:

```
pip install notebook
```

Download MACRO

Note: The following steps assume that you have a GitHub account and that you have already created a token to download the repository. To create a new token, you can go to your GitHub settings and click on "Generate new token". Please refer to the GitHub documentation for more information on how to create a token.

Since MACRO is a **private repository**, you need to have permissions to clone the repository. Once your user has been added to the repository, to clone the repository you can use the following command in your terminal:

```
git clone -b <branch-name> https://<username>:<token>@github.com/macroenergy/Macro.git
```

Alternatively, you can setup an SSH key pair and use the SSH URL instead of the HTTPS URL.

Installation steps

- **Navigate to the repository**:

```
cd Macro
```

- **Install MACRO and all the dependencies**:

```
julia --project -e 'using Pkg; Pkg.instantiate(); Pkg.precompile()'
```

Setting up Jupyter Notebook

Once MACRO is installed, to enable Jupyter Notebook support, you can run the following command:

```
julia --project -e 'using IJulia; IJulia.installkernel("Macro", "--project=@.")'
```

Once the kernel is installed, you can run Jupyter Notebook with one of the following commands:

```
jupyter lab
```

or

```
jupyter notebook
```

Testing the installation

To test the installation, you can run the following command:

```julia
using Macro
```

in a Jupyter Notebook cell or in a Julia terminal. If everything is set up correctly, you should see no errors and the package should load without any issues.

Running these notebooks on GitHub Codespaces (optional)

One simple way to run the notebooks is by using GitHub Codespaces.

Note: A GitHub Codespace is a cloud-based development environment that is hosted on GitHub's infrastructure. Therefore, a GitHub account is required to access it.

The repository is already configured to be used with GitHub Codespaces, and the following steps will guide you through the process:

1. Navigate to the repository on GitHub MACRO.

2. Change the branch to `tutorials`.
3. Click on the "Code" button and then on "Codespaces".
4. (Optional) GitHub allows you to configure the codespace to use a specific hardware. To do that, click on the three dots on the top right corner of the pop-up dialog and then click on "New with options".

Note: All personal GitHub accounts are limited to 120 hours of compute time and 15GB of storage per month. You can learn more about the limitations here and here.

1. If you want to use the default hardware, skip the previous step and click on "Create codespace on tutorials".
2. Once the codespace is open, remember to install the dependencies of MACRO by copying and pasting the following command in the terminal at the bottom of the page:

```
julia --project -e 'using Pkg; Pkg.instantiate(); Pkg.precompile()'
```

1. Once the dependencies are installed, you can run the notebooks by either opening the `.ipynb` files in the `tutorials` folder or by copying and pasting the following command in the terminal at the bottom of the page:

```
make run-notebooks
```

The last command will open a Jupyterlab instance in the browser. Once the Jupyterlab is open, you can run the notebooks by clicking on the "Open in Browser" button that should appear on the bottom left corner of the screen. If a token is required to access the notebook, you can find it in the terminal where you ran the `make run-notebooks` command. The token is usually located at the end of the URL shown in the terminal. For example:

```
http://<codespace-name>-<username>.app.github.dev/lab?token=<token>
```

Simply copy and paste the token in the box that appears in the browser. Alternatively, you can simply copy and paste the full URL in the browser's address bar.

If you also click on the "Make Public" button, you will be able to share the notebook with others over the web.

Hint: Speed up the build time of the codespace with a prebuild

To speed up the build time of the codespace, you can fork the repository and then setup a **Prebuild** for your fork. This will allow you to avoid to re-install all the dependencies every time you open a new codespace. To setup a prebuild:

1. Fork the repository.
2. Go to the **Settings** of your fork and then click on **Codespaces** under the **Code and automation** section.

3. Click on **Setup prebuild**.
4. Choose the correct branch (e.g. `tutorials`).
5. Select if you want to prebuild the codespace on **Every push** to the branch or **On schedule**.
6. Select the **Region availability** for your prebuild.
7. Click on **Create** to finish the setup.

Once the prebuild is created, every time you open a new codespace from your fork, it will use the prebuild to speed up the build time.

Chapter 5

Input Files

5.1 MACRO input files

Interactive Notebook

The interactive version of this tutorial can be found here.

To configure a MACRO case, the user needs to provide a set of input files describing at least the following components:

- **System configuration**
- **Assets/technologies**
- **Model settings**

Remove comments from JSON files

The comments in the JSON files below are only for the user's reference and should be removed before using them as input for a MACRO case.

System configuration

The system configuration files are located in the `system` folder and include the following five files:

- `commodities.json`: describes the commodities flowing through the system.
- `time_data.json`: describes the temporal resolution for each sector in the simulation.
- `nodes.json`: describes the nodes in the system.
- `demand.csv`: describes the demand for each commodity at each node in the system.
- `fuel_prices.csv`: (`supply_data`) describes the fuel prices for each commodity at each node in the system.

`commodities.json`

The `commodities.json` file is a JSON file with a list of commodities to include in the case. This is how the file looks like:

```
s = read("one_zone_electricity_only/system/commodities.json", String)
println(s)
```

To include new commodities, the user just needs to add the new commodity to the list of commodities in the `commodities.json` file.

`time_data.json`

This file describes the temporal resolution of the simulation. It contains three main parameters:

- `PeriodLength`: total number of hours in the simulation.
- `HoursPerTimeStep`: number of hours in each time step.
- `HoursPerSubperiod`: number of hours in each subperiod, where a subperiod represents a time slice of the simulation used to perform time wrapping for time-coupling constraints (see, for example, Macro.timestep-before).

Except for the `PeriodLength`, all the other parameters need to be provided for each commodity in the system. For example, the following is a complete example of the `time_data.json` file:

```
{
    "PeriodLength": 8760,   // units: hours
    "HoursPerTimeStep": {
        "Electricity": 1,  // units: hours
        "NaturalGas": 1,
        "CO2": 1
    },
    "HoursPerSubperiod": {
        "Electricity": 8760,  // units: hours
        "NaturalGas": 8760,
        "CO2": 8760
    }
}
```

In the example above, the simulation will run for 8760 hours (one year), with one hour per time step and one hour per subperiod for each commodity (single subperiod).

Units

In MACRO, everything that is transformed into electricity is expressed in MWh, and all the other commodities are expressed in metric tons:

- **Time**: hours
- **Electricity**: MWh
- **NaturalGas**: MWh

- **Coal**: MWh
- **Uranium**: MWh
- **Hydrogen**: MWh
- **CO2**: tonnes
- **CO2Captured**: tonnes
- **Biomass**: tonnes

`nodes.json`

The `nodes.json` file is a JSON file with a list of nodes to include in the case. The file is structured as follows:

1. Each node type (like "NaturalGas") is defined as a dictionary with three main components:
   - `type`: The category of the node (e.g., "NaturalGas")
   - `global_data`: attributes that apply to all instances of this node type
     - `time_interval`: time resolution for the time series in the node (needs to match one of the commodities in the `time_data.json` file)
   - `instance_data`: A list of specific nodes of this type, each with:
     - `id`: A unique identifier for the node

All other fields (e.g, `demand`, `price`, `constraints`, etc.) are optional and will take the default values if not provided. A complete list of the fields that can be included in the `nodes.json` file can be found here `Macro.Node`(@ref).

Here's a simplified example:

```
"nodes": [
    {
        "type": "NaturalGas",
        "global_data": {
            "time_interval": "NaturalGas"
        },
        "instance_data": [
            {
                "id": "natgas_1",
                "price": {
                    "timeseries": {
                        "path": "system/fuel_prices.csv",
                        "header": "natgas_1"
                    }
                }
            },
            {
                "id": "natgas_2",
                "price": {
                    "timeseries": {
                        "path": "system/fuel_prices.csv",
                        "header": "natgas_2"
                    }
                }
            }
            // ... more instances ...
        ]
    }
]
```

This structure allows you to define multiple nodes of the same type while sharing common settings through `global_data`.

The following is a slightly more complete example of how to include nodes of type "Electricity", containing the electricity demand at each node in the system:

```
{
    "type": "Electricity",
    "global_data": {
        "time_interval": "Electricity",
        "constraints": {
            "BalanceConstraint": true,
        }
    },
    "instance_data": [
        {
            "id": "elec_SE",
            "demand": {
                "timeseries": {
                    "path": "system/demand.csv",
                    "header": "Demand_MW_z1"
                }
            }
        },
        {
            "id": "elec_MIDAT",
            "demand": {
                "timeseries": {
                    "path": "system/demand.csv",
                    "header": "Demand_MW_z2"
                }
            }
        },
        {
            "id": "elec_NE",
            "demand": {
                "timeseries": {
                    "path": "system/demand.csv",
                    "header": "Demand_MW_z3"
                }
            }
        }
    ]
}
```

As mentioned before, **Demand** and **price** time series are linked to columns in the demand and `fuel_prices` files respectively. The user needs to make sure that the **path** matches the file location and the **header** matches the column name in the file. In the case above, for example, the node `elec_SE` is linked to the column Demand_MW_z1 in the demand.csv file, the node elec_MIDAT to the column Demand_MW_z2, and the node elec_NE to the column Demand_MW_z3. In particular, the demand.csv file has the following structure:

```
using CSV, DataFrames
demand_data = CSV.read("one_zone_electricity_only/system/demand.csv", DataFrame)
first(demand_data, 10)
```

Constraints

One of the main features of MACRO is the ability to include constraints on the system from a pre-defined library of constraints. To include a constraint to a node, the user needs to add the constraint name to the constraints dictionary in the node's global_data or instance_data field.

For example, to include the BalanceConstraint to the node elec_SE, the user needs to add the following to the nodes.json file:

```
"constraints": {
    "BalanceConstraint": true
}
```

With the BalanceConstraint, the model will ensure that the sum of the supply and demand at each node and each time step is zero.

Another example is the MaxNonServedDemandConstraint, which limits the maximum amount of demand that can be unmet in a given time step. To include this constraint to the node elec_SE, the user needs to add the following to the nodes.json file:

```
"max_nds": [1],  // units: fraction of the demand
"price_nsd": [5000.0], // units: $/MWh
"constraints": {
    "MaxNonServedDemandConstraint": true,
    "MaxNonServedDemandPerSegmentConstraint": true
}
```

This will add the MaxNonServedDemandConstraint and the MaxNonServedDemandPerSegmentConstraint to the node elec_SE with a maximum non-served demand equal to the demand in each period, and a price of 5000.0 $/MWh for the unmet demand ("max_nds" is the fraction of the demand that can be unmet).

A complete list of the constraints available in MACRO can be found in the MACRO Constraint Library section.

Therefore, a complete example of the nodes.json file for the electricity system is the following:

```
{
    "type": "Electricity",
    "global_data": {
        "time_interval": "Electricity",
        "max_nsd": [
            1   // units: fraction of the demand
        ],
        "price_nsd": [
            5000.0 // units: $/MWh
        ],
        "constraints": {
            "BalanceConstraint": true,
            "MaxNonServedDemandConstraint": true,
            "MaxNonServedDemandPerSegmentConstraint": true
        }
    },
    "instance_data": [
        {
            "id": "elec_SE",
            "demand": {
                "timeseries": {
                    "path": "system/demand.csv",
                    "header": "Demand_MW_z1"
                }
            }
        },
        {
            "id": "elec_MIDAT",
            "demand": {
                "timeseries": {
                    "path": "system/demand.csv",
                    "header": "Demand_MW_z2"
                }
            }
        },
        {
            "id": "elec_NE",
            "demand": {
                "timeseries": {
                    "path": "system/demand.csv",
                    "header": "Demand_MW_z3"
                }
            }
        }
    ]
}
```

Time series data

The time series data for the demand and fuel prices are CSV files with the same format:

- First column: time step
- Second column and following: value of the time series for each time step

For example, the `demand.csv` file has the following structure:

```julia
using CSV, DataFrames
demand_data = CSV.read("one_zone_electricity_only/system/demand.csv", DataFrame)
first(demand_data, 10)
```

And the `fuel_prices.csv` file has a similar structure:

```
fuel_prices_data = CSV.read("one_zone_electricity_only/system/fuel_prices.csv", DataFrame)
first(fuel_prices_data, 10)
```

where the names of the columns must match the `header` field in the `nodes.json` file for the corresponding node.

The units are `USD/MWh` of natural gas.

Assets/technologies

The assets folder contains all the input files for the assets/resources as well as transmission lines and storage technologies that are included in the system.

Each asset or technology is defined in a separate file, and all files share a similar structure, which closely resembles the structure of the nodes in the `nodes.json` file. Each asset or technology includes the following fields:

- `type`: The type of the asset or technology.
- `global_data`: Attributes that apply universally to all instances of this asset or technology.
- `instance_data`: A list of specific instances of this asset or technology, where each instance contains:
  - `id`: A unique identifier for the asset or technology.

Similarly to the nodes, all the fields defined in the `global_data` will be shared by all the instances of the asset/technology, and the fields defined in `instance_data` can be different for each instance.

Note: To understand the structure of the attributes in both `global_data` and `instance_data`, the user can refer to the asset definitions in the MACRO Asset Library section.

To familiarize with the structure of the asset/technology input files, let's now see an example of a **system** with:

- a natural gas power plant,
- a solar photovoltaic pannel,
- a wind turbine,
- a battery.

Natural gas power plant

Type representation

The type that defines the natural gas power plant is `ThermalPower` and can be found in `src/model/assets/thermalpower.jl`. Here is a schematic representation of the type:

As expected, a natural gas power plant is made of the followning components:

- **ng → elec transformation**: a transformation object that defines the conversion of the fuel to electricity
- three edges that connect the asset to the rest of the system:
  - **fuel edge**: natural gas input

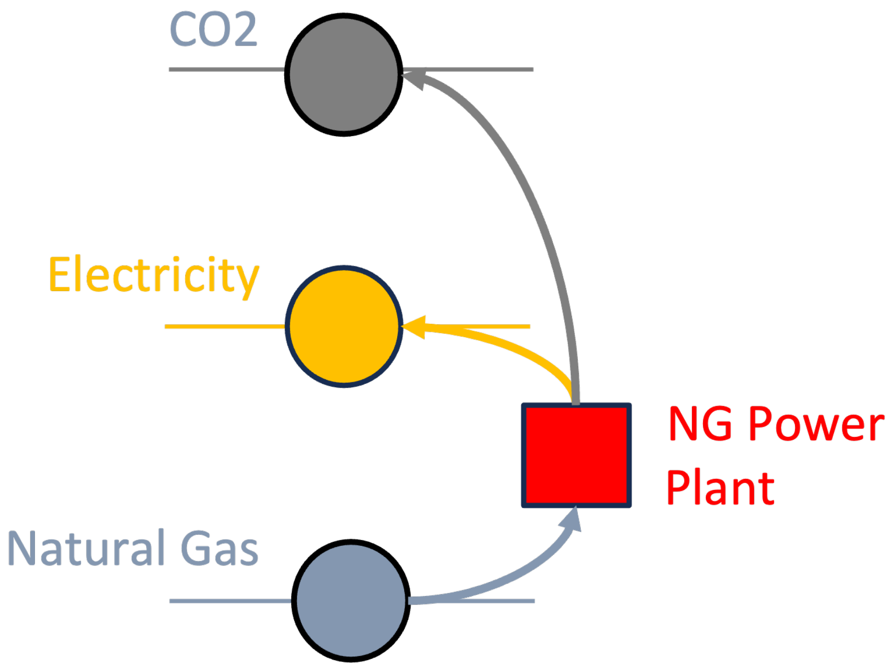

Figure 5.1: ThermalPower

- **electricity edge**: electricity output
- **co2 edge**: CO2 output

Keeping the type representation in mind, we can now see how the asset is defined in the `src/model/assets/thermalpower.jl` file:

```
struct ThermalPower{T} <: AbstractAsset
    id::AssetId
    thermal_transform::Transformation
    elec_edge::Union{Edge{Electricity},EdgeWithUC{Electricity}}
    fuel_edge::Edge{T}
    co2_edge::Edge{CO2}
end
```

In the object above, the `id` field is a unique identifier for the asset/technology, and the rest of the fields are the implementation of the transformation and the three edges that we saw before in the type representation.

Asset input file

The input file for the natural gas power plant is `one_zone/assets/naturalgas_power.json`. If we open the file, we can notice that, apart from the `id` field, the rest of the fields match the ones we saw in the type representation:

```
{
    "transforms":{
        // ... thermal_transform fields ...
    },
    "edges":{
        "elec_edge": {
            // ... elec_edge fields ...
        },
        "fuel_edge": {
            // ... fuel_edge fields ...
        },
        "co2_edge": {
            // ... co2_edge fields ...
        }
    }
}
```

Some important attributes for the `transforms` field are:

- `timedata`: time resolution for the time series in the transformation (needs to match one of the commodities in the `time_data.json` file)
- `constraints`: constraints to be applied to the transformation
- `emission_rate`: emission rate of the transformation (tons of CO2/MWh of natural gas)
- `efficiency_rate`: efficiency rate of the transformation (MWh of electricity/MWh of natural gas)

For the edges field, some important attributes are:

- `type`: type of the commodity in the edge
- `start_vertex`: start vertex of the edge
- `end_vertex`: end vertex of the edge
- `unidirectional`: whether the edge is unidirectional or bidirectional
- `has_capacity`: whether the edge can expand during the optimization
- `uc`: whether the edge is a unit commitment variable
- `constraints`: constraints to be applied to the edge
- `investment_cost/fixed_om_cost/variable_om_cost`: investment, fixed, and variable operating costs for the asset
- `existing_capacity`: existing capacity of the asset

The complete list of fields that can be included in each of the fields `transforms`, `elec_edge`, `fuel_edge`, and `co2_edge` can be found in the definition of the `Edge` type in the `Macro.Edge`(@ref), `Macro.EdgeWithUC`(@ref), and `Transformation` type in the `Macro.Transformation`(@ref) files.

For example, the following is a complete example of the `naturalgas_power.json` file:

```
{
    "NaturalGasPower": [
        {
            "type": "ThermalPower",
            "global_data": {
                "transforms": {
                    "timedata": "NaturalGas",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges" : {
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": true,
                        "uc": true,
                        "integer_decisions": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true,
                            "MinFlowConstraint": true,
                            "MinUpTimeConstraint": true,
                            "MinDownTimeConstraint": true
                        }
                    },
                    "fuel_edge": {
                        "type": "NaturalGas",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_sink"
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_naturalgas_ccavgcf_moderate_0",
                    "transforms":{
                        "emission_rate": 0.181048235160161, // units: tons of CO2/MWh of natural
                        ↪  gas
                        "efficiency_rate": 0.53624731 // units: MWh of electricity/MWh of natural
                        ↪  gas
                    },
                    "edges":{
                        "elec_edge": {
                            "end_vertex": "elec_SE",
                            "can_retire": true,
                            "can_expand": true,
                            "existing_capacity": 0.0,
                            "investment_cost": 78218.72932, // units: $/MW/year (annualized)
```

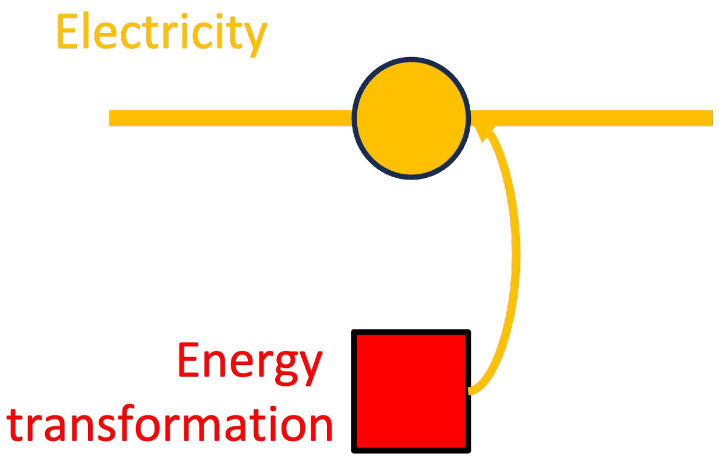

Figure 5.2: Vre

```
                            "fixed_om_cost": 27300, // units: $/MW/year (annualized)
                            "variable_om_cost": 1.74, // units: $/MWh
                            "capacity_size": 573,
                            "startup_cost": 61, // units: $/MW per start
                            "startup_fuel": 0.058614214,  // units: MWh/MW of capacity per start
                            "min_up_time": 4, // units: hours
                            "min_down_time": 4, // units: hours
                            "ramp_up_fraction": 1, // units: fraction of the capacity
                            "ramp_down_fraction": 1, // units: fraction of the capacity
                            "min_flow_fraction": 0.3 // units: fraction of the capacity
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_SE"
                        }
                    }
                }
            ]
        }
    ]
}
```

Remove comments from JSON files

The comments in the JSON file above are only for the user's reference and should be removed before using them as input for a MACRO case.

Solar photovoltaic pannel and wind turbine

Since the solar photovoltaic pannel and the wind turbine are very similar, the type that defines them is called `Vre` and can be found in `src/model/assets/vre.jl`. Here is a schematic representation of the type:

As can be seen, a solar photovoltaic pannel and a wind turbine are made of the following components:

- **solar/wind energy → electricity transformation**: a transformation object that defines the conversion of the energy to electricity
- single edge that connects the asset to the rest of the system:
  - **electricity edge**: electricity output

The asset is therefore defined as follows:

```
struct VRE <: AbstractAsset
    id::AssetId
    energy_transform::Transformation
    elec_edge::Edge{Electricity}
end
```

The input file for variable renewable resources is `one_zone/assets/solar_pv.json`. If we open the file, we can notice the usual structure for nodes and assets that we should be familiar with by now:

- **type**: type of the asset/technology
- **global_data**: settings that apply to all instances of this asset/technology
- **instance_data**: a list of specific assets/technologies of this type, each with:
  - `id`: a unique identifier for the asset/technology

In the `global_data` and `instance_data` fields, we can notice that we have a `transforms` field that contains the attributes for the `energy_transform` object, and an edges field that contains the attributes for the edge object that connects the asset to the electricity network:

```
"transforms": {
    // ... energy_transform fields ...
}
"edges": {
    // ... edge fields ...
}
```

For example, the following is a complete example of the `vre.json` file that defines a solar pv and a wind turbine:

```
{
    "vre": [
        {
            "type": "VRE",
            "global_data": {
                "transforms": {
                    "timedata": "Electricity"
                },
                "edges": {
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "can_expand": true,
                        "can_retire": false,
                        "has_capacity": true,
                        "constraints": {
                            "CapacityConstraint": true
                        }
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_utilitypv_class1_moderate_70_0_2_1",
                    "edges": {
                        "elec_edge": {
                            "fixed_om_cost": 15390.48615,
                            "investment_cost": 49950.17548,
                            "end_vertex": "elec_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_utilitypv_class1_moderate_70_0_2_1"
                                }
                            }
                        }
                    }
                },
                {
                    "id": "SE_landbasedwind_class4_moderate_70_1",
                    "edges": {
                        "elec_edge": {
                            "fixed_om_cost": 34568.125,
                            "investment_cost": 68099.00274,
                            "end_vertex": "elec_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_landbasedwind_class4_moderate_70_1"
                                }
                            }
                        }
                    },
                }
            ]
        }
    ]
}
```

Something to notice is that the `availability` field is a time series that needs to be provided for each instance of the VREasset. This time series describes the availability of the resource at each time step. This attribute looks very similar to the demand field in the nodes input file, where the `path` field is used to link to the time series data in a csv file, and the `header` field is used to match the column name in the file.

This is an example of the `availability.csv` file:

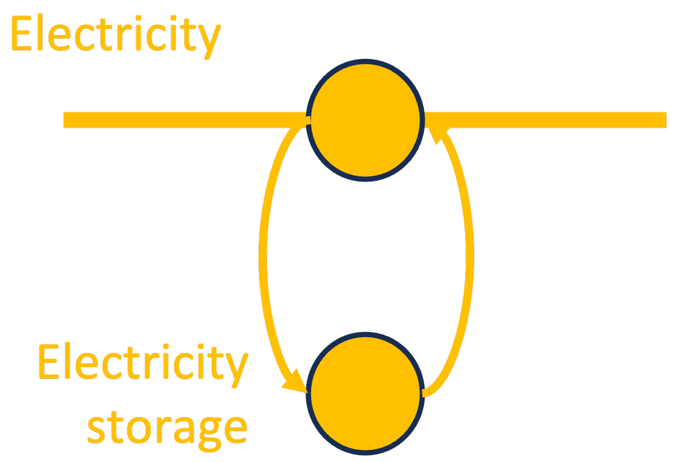

Figure 5.3: Battery

```
availability_data = CSV.read("one_zone_electricity_only/assets/availability.csv", DataFrame)
first(availability_data, 10)
```

The units are `fraction of installed capacity`.

```
using Pkg; Pkg.add("Plots")
using Plots
p = Plots.plot(availability_data[:,1], availability_data[:,2], label="Solar PV", xlabel="Time (h)",
↪  ylabel="Availability")
Plots.plot!(p, availability_data[:,1], availability_data[:,3], label="Wind")
```

Battery (Electricity)

Finally, let's see how storage technologies are defined in MACRO and how to include them in the system.

A battery is defined by the following components:

- `battery_storage`
- `charge_edge`: a charge edge that connects the battery to the electricity network
- `discharge_edge`: a discharge edge that connects the battery to the electricity network

The input file for the battery is `one_zone/assets/battery.json`. Again, if we open the file, we can recognize the usual structure for nodes and assets:

- **type**: type of the asset/technology
- **global_data**: settings that apply to all instances of this asset/technology
- **instance_data**: a list of specific assets/technologies of this type, each with:

  - `id`: a unique identifier for the asset/technology

This is an example of the `battery.json` file: “‘json { "elec*stor": [{ "type": "Battery", "global*data": { "edges": { "discharge*edge": { "type": "Electricity", "unidirectional": true, "has*capacity": true, "can*expand": true, "can*retire": false, "constraints": { "CapacityConstraint": true, "StorageDischargeLimitConstraint": true } }, "charge*edge": { "type": "Electricity", "unidirectional": true, "has*capacity": false } }, "storage": { "commodity": "Electricity", "can*expand": true, "can*retire": false, "constraints": { "StorageCapacityConstraint": true, "StorageSymmetricCapacityConstraint": true, "BalanceConstraint": true } } }, "instance*data": [{ "id": "battery*SE", "edges": { "discharge*edge": { "end*vertex": "elec*SE", "existing*capacity" : 0.0, "fixed*om*cost" : 4536.98, "investment*cost": 17239.56121, "variable*om*cost": 0.15, "efficiency": 0.92 }, "charge*edge": { "start*vertex": "elec*SE", "efficiency": 0.92, "variable*om*cost": 0.15 } }, "storage":{ "existing*capacity*storage": 0.0, "fixed*om*cost*storage": 2541.19, // units: USD/MWh-year (annualized) "investment*cost*storage": 9656.002735, // units: USD/MWh-year (annualized) "max*duration": 10, // units: hours "min_duration": 1 // units: hours } }] }] }

Something to notice is that both the edges and `storage` fields contain a `constraints` field. This is an example of a constraint that can be applied to a battery:

- `StorageCapacityConstraint`: ensures that the capacity of the battery is not exceeded
- `StorageSymmetricCapacityConstraint`: ensures that simultaneous charge and discharge do not exceed the capacity of the battery
- `BalanceConstraint`: ensures that the battery contributes to the demand balance of the system

Model settings

This folder contains the `macro_settings.json` file, which is used to set the model settings and tune the model. It currently contains only one setting:

- `ScalingFactor = true/false`: if true, the model will scale the input data to the following units: MWh → GWh, tons → ktons, $\/MWh → M$\/GWh, $\/ton → M$\/kton

Summary

In this tutorial, we have seen how to define a system in MACRO by providing the input files in the `system` and `assets` folders.

- Almost all the files except for time series data are JSON files, which allow for a high degree of customization of the system.
- Time series data are provided in CSV files, where the first column is the time step and the following columns are the time series for each node/asset.
- Both nodes and assets have a similar structure:
  - `type`: type of the node/asset
  - `global_data`: settings that apply to all instances of the node/asset
  - `instance_data`: a list of specific nodes/assets of this type, each with:
    - `id`: a unique identifier for the node/asset

  which allows to define multiple instances of the same node/asset type while sharing common settings through the `global_data` field.

- Each asset is defined by a set of edges, transformations, and storage objects, which are defined in the `edges`, `transforms`, and `storage` fields respectively.
- All the fields in the `global_data` and `instance_data` fields can be customized, and the complete list of fields that can be included in each of the fields `transforms`, `edges`, and `storage` can be found in the definition of the `Edge`, `Transformation`, and `Storage` types in the `Macro.Edge`(@ref), `Macro.Transformation`(@ref), and `Macro.Storage`(@ref) files.
- All the constraints available in MACRO can be found in the MACRO Constraint Library section.
- All the assets available in MACRO can be found in the MACRO Asset Library section, where each file contains the definition of the asset type.

Chapter 6

Running Macro

6.1 Running MACRO

Interactive Notebook

The interactive version of this tutorial can be found here.

In this tutorial, we start from a single zone electricity system with four resource clusters: utility scale solar PV, land-based wind power generation, natural gas combined cycle power plants, and electricity storage.

We consider three commodities: electricity, natural gas, and CO_2.

Initially, hydrogen is modeled exogenously, adding a constant electricity demand for hydrogen production to the electricity demand time series. In other words, we assume the existance of an electrolyzer that continuously consumes electricity to meet the hydrogen demand.

We model a greenfield scenario with a carbon price of 200 USD/ton, i.e., we allow CO_2 emissions with a penalty cost.

Note: We use the default units in MACRO: MWh for energy vectors, metric tons for other commodities (e.g., CO_2) and dollars for costs

```
using Pkg; Pkg.add("VegaLite")
```

```
using Macro
using HiGHS
using CSV
using DataFrames
using JSON3
using VegaLite
```

We first load the inputs:

```
system = Macro.load_system("one_zone_electricity_only");
```

We are now ready to generate the MACRO capacity expansion model. Because MACRO is designed to be solved by high performance decomposition algorithms, the model formulation has a specific block structure that can be exploited by these schemes. In the case of 3 operational sub-periods, the block structure looks like this:

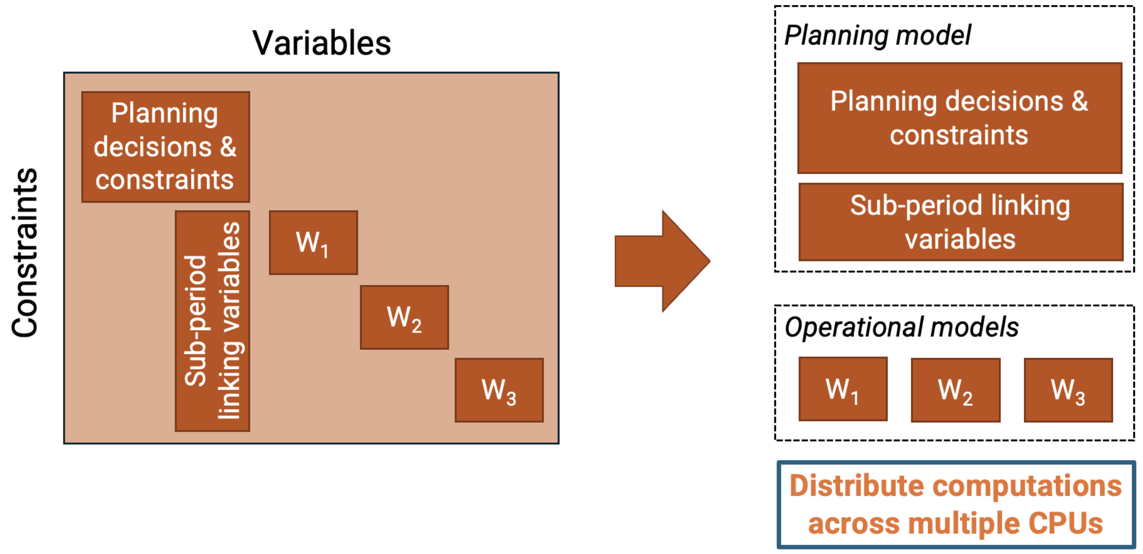
Variables
Constraints
Planning decisions & constraints
Sub-period linking variables
W$_1$
W$_2$
W$_3$
Planning model
Planning decisions & constraints
Sub-period linking variables
Operational models
W$_1$
W$_2$
W$_3$
Distribute computations across multiple CPUs

Figure 6.1: model_structure

```
model = Macro.generate_model(system)
```

Next, we set the optimizer. Note that we are using the open-source LP solver HiGHS, alternatives include the commerical solvers Gurobi, CPLEX, COPT.

```
Macro.set_optimizer(model, HiGHS.Optimizer);
```

Finally, we solve the capacity expansion model:

```
Macro.optimize!(model)
```

And extract the results:

```
capacity_results = Macro.get_optimal_asset_capacity(system)
```

The total system cost (in dollars) is:

```
Macro.objective_value(model)
```

and the total emissions (in metric tonnes) are:

```
co2_node = Macro.get_nodes_sametype(system.locations, CO2)[1]
Macro.value(sum(co2_node.operation_expr[:emissions]))
```

We can also plot the electricity generation results using `VegaLite.jl`:

```
plot_time_interval = 3600:3624
natgas_power = Macro.value.(Macro.flow(system.assets[2].elec_edge)).data[plot_time_interval] / 1e3;
solar_power = Macro.value.(Macro.flow(system.assets[3].edge)).data[plot_time_interval] / 1e3;
wind_power = Macro.value.(Macro.flow(system.assets[4].edge)).data[plot_time_interval] / 1e3;

elec_gen = DataFrame(hours=plot_time_interval,
    solar_photovoltaic=solar_power,
    wind_turbine=wind_power,
    natural_gas_fired_combined_cycle=natgas_power,
)

stack_elec_gen = stack(elec_gen, [:natural_gas_fired_combined_cycle, :wind_turbine,
↪  :solar_photovoltaic], variable_name=:resource, value_name=:generation);

elc_plot = stack_elec_gen |>
           @vlplot(
    :area,
    x = {:hours, title = "Hours"},
    y = {:generation, title = "Electricity generation (GWh)", stack = :zero},
    color = {"resource:n", scale = {scheme = :category10}},
    width = 400,
    height = 300
)
```

Exercise 1

Set a strict net-zero CO_2 cap by removing the slack allowing constraint violation for a penalty. This can be done by deleting the field `price_unmet_policy` from the CO_2 node in file `one_zone_electricity_only/system/nodes.json`

Then, re-run the model with these new inputs and show the capacity results, total system cost, emissions, and plot the generation profiles.

Solution

Open file `one_zone_electricity_only/system/nodes.json`, go to the bottom of the file where the CO_2 node is defined. Remove the lines related to the field `price_unmet_policy`, so that the node definition looks like this:

```
 {
    "type": "CO2",
    "global_data": {
        "time_interval": "CO2"
    },
    "instance_data": [
        {
            "id": "co2_sink",
            "constraints": {
                "CO2CapConstraint": true
            },
            "rhs_policy": {
                "CO2CapConstraint": 0
            }
        }
    ]
}
```

Then, you need to re-load the inputs:

```
system = Macro.load_system("one_zone_electricity_only");
```

generate the MACRO model:

```
model = Macro.generate_model(system);
```

and solve it:

```
Macro.set_optimizer(model, HiGHS.Optimizer);
Macro.optimize!(model)
```

We can check the results by printing the total system cost:

```
Macro.objective_value(model)
```

and the new emissions (which should be zero):

```
co2_node = Macro.get_nodes_sametype(system.locations, CO2)[1]
Macro.value(sum(co2_node.operation_expr[:emissions]))
```

Finally, we plot the generation results:

```
plot_time_interval = 3600:3624
natgas_power =  Macro.value.(Macro.flow(system.assets[2].elec_edge)).data[plot_time_interval]/1e3;
solar_power = Macro.value.(Macro.flow(system.assets[3].edge)).data[plot_time_interval]/1e3;
wind_power = Macro.value.(Macro.flow(system.assets[4].edge)).data[plot_time_interval]/1e3;

elec_gen =  DataFrame( hours = plot_time_interval,
                solar_photovoltaic = solar_power,
                wind_turbine = wind_power,
                natural_gas_fired_combined_cycle = natgas_power,
                )

stack_elec_gen = stack(elec_gen,
↪   [:natural_gas_fired_combined_cycle,:wind_turbine,:solar_photovoltaic], variable_name=:resource,
↪   value_name=:generation);

elc_plot = stack_elec_gen |>
@vlplot(
    :area,
    x={:hours, title="Hours"},
    y={:generation, title="Electricity generation (GWh)",stack=:zero},
    color={"resource:n", scale={scheme=:category10}},
    width=400,
    height=300
)
```

Chapter 7

Multisector Modelling

7.1 Multisector modelling with MACRO

Interactive Notebook

The interactive version of this tutorial can be found here.

In this tutorial, we extend the electricity-only model considered in Tutorial 2 to build a multisector model for joint capacity expansion in electricity and hydrogen sectors.

To do this, we scorporate hydrogen and electricity demand from Tutorial 2, and endogeneously model hydrogen production and storage in MACRO.

```
using Pkg; Pkg.add(["VegaLite", "Plots"])
```

```
using Macro
using HiGHS
using CSV
using DataFrames
using JSON3
using Plots
using VegaLite
```

Create a new case folder named "one*zone*multisector"

```
if !isdir("one_zone_multisector")
    mkdir("one_zone_multisector")
    cp("one_zone_electricity_only/assets","one_zone_multisector/assets", force=true)
    cp("one_zone_electricity_only/settings","one_zone_multisector/settings", force=true)
    cp("one_zone_electricity_only/system","one_zone_multisector/system", force=true)
    cp("one_zone_electricity_only/system_data.json","one_zone_multisector/system_data.json",
    ↪  force=true)
end
```

Note: If you have previously run Tutorial 2, make sure that file `one_zone_multisector/system/nodes.json` is restored to the original version with a CO_2 price. The definition of the CO_2 node should look like this:

```
{
    "type": "CO2",
    "global_data": {
        "time_interval": "CO2"
    },
    "instance_data": [
        {
            "id": "co2_sink",
            "constraints": {
                "CO2CapConstraint": true
            },
            "rhs_policy": {
                "CO2CapConstraint": 0
            },
            "price_unmet_policy":{
                "CO2CapConstraint": 200
            }
        }
    ]
}
```

Add Hydrogen to the list of modeled commodities, modifying file one_zone_multisector/system/commodities.json:

```
new_macro_commodities = Dict("commodities"=> ["Electricity", "NaturalGas", "CO2", "Hydrogen"])

open("one_zone_multisector/system/commodities.json", "w") do io
    JSON3.pretty(io, new_macro_commodities)
end
```

Update file one_zone_multisector/system/time_data.json accordingly:

```
new_time_data = Dict("PeriodLength"=>8760,
new_time_data = Dict("PeriodLength"=>8760,
                    "HoursPerTimeStep"=>Dict("Electricity"=>1, "NaturalGas"=> 1, "CO2"=> 1,
                    ↪  "Hydrogen"=>1),
                    "HoursPerSubperiod"=>Dict("Electricity"=>8760, "NaturalGas"=> 8760, "CO2"=>
                    ↪  8760, "Hydrogen"=>8760)
                )

open("one_zone_multisector/system/time_data.json", "w") do io
    JSON3.pretty(io, new_time_data)
end
```

Move separate electricity and hydrogen demand timeseries into the system folder

```
cp("demand_timeseries/electricity_demand.csv","one_zone_multisector/system/demand.csv";force=true)
```

```
cp("demand_timeseries/hydrogen_demand.csv","one_zone_multisector/system/hydrogen_demand.csv";force=true)
```

Exercise 1

using the existing electricity nodes in `one_zone_multisector/system/nodes.json` as template, add an Hydrogen demand node, linking it to the `hydogen_demand.csv` timeseries.

Solution

The definition of the new Hydrogen node in `one_zone_multisector/system/nodes.json` should look like this:

```
    {
        "type": "Hydrogen",
        "global_data": {
            "time_interval": "Hydrogen",
            "constraints": {
                "BalanceConstraint": true
            }
        },
        "instance_data": [
            {
                "id": "h2_SE",
                "demand": {
                    "timeseries": {
                        "path": "system/hydrogen_demand.csv",
                        "header": "Demand_H2_z1"
                    }
                }
            }
        ]
    },
```

Next, add an electrolyzer asset represented in MACRO as a transformation connecting electricity and hydrogen nodes:

To include the electrolyzer, create a file `one_zone_multisector/assets/electrolyzer.json` based on the asset definition in `src/model/assets/electrolyzer.jl`:

```
{
   "electrolyzer": [
        {
            "type": "Electrolyzer",
            "global_data":{
                "transforms": {
                    "timedata": "Electricity",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "h2_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_retire": true,
                        "can_expand": true,
                        "constraints": {
```

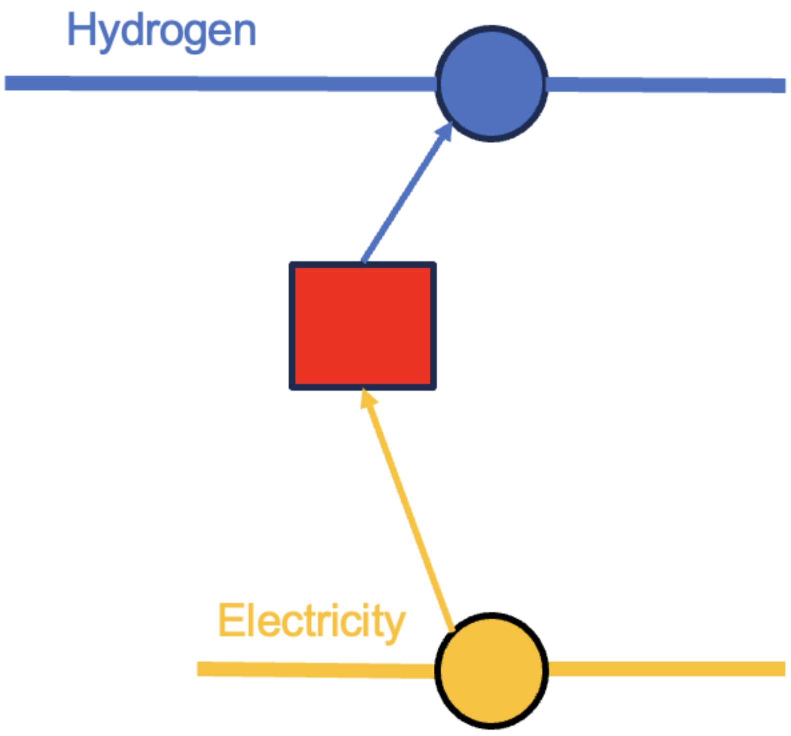

Figure 7.1: electrolyzer

```
                        "CapacityConstraint": true,
                        "RampingLimitConstraint": true,
                        "MinFlowConstraint": true
                    }
                },
                "elec_edge": {
                    "type": "Electricity",
                    "unidirectional": true,
                    "has_capacity": false
                }
            }
        },
        "instance_data":[
            {
                "id": "SE_Electrolyzer",
                "transforms":{
                    "efficiency_rate": 0.875111139 // units: # MWh of H2 / MWh of electricity
                },
                "edges":{
                    "elec_edge": {
                        "start_vertex": "elec_SE"
                    },
                    "h2_edge": {
                        "end_vertex": "h2_SE",
                        "existing_capacity": 0,
```

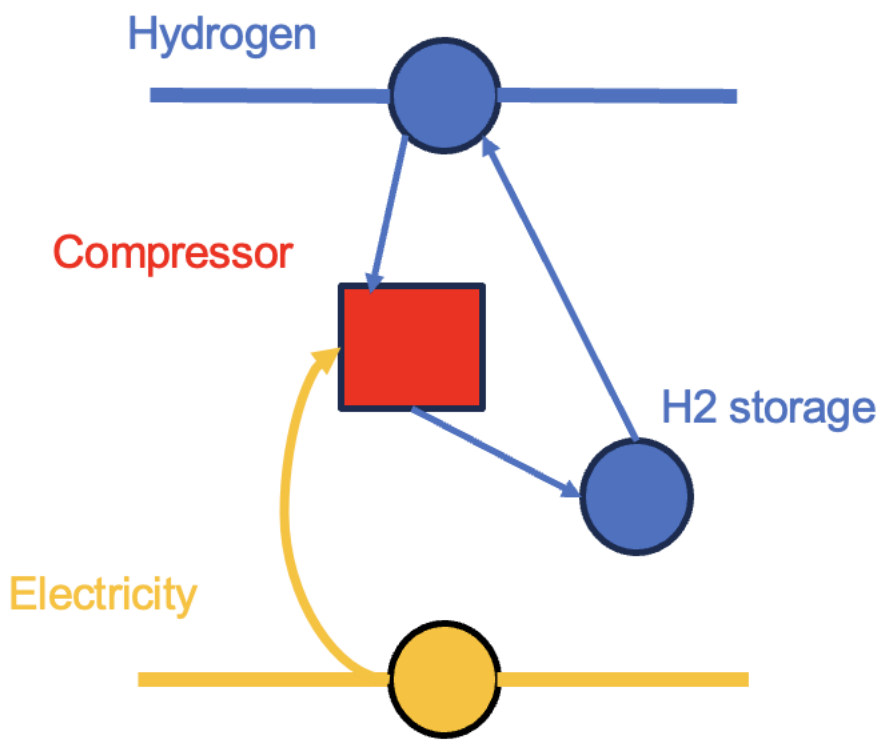

Figure 7.2: hydrogen_storage

```
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction":0.1
                        }
                    }
                }
            ]
        }
    ]
}
```

Include an hydrogen storage resource cluster, represented in MACRO as combination of a compressor transformation (consuming electricity to compress the gas) and a storage node:

Add a file `one_zone_multisector/assets/h2_storage.json` based on the asset definition in `src/model/assets/gasstorage.jl`
should look like this:

```
{
    "h2stor": [
        {
            "type": "GasStorage",
            "global_data": {
                "nodes": {},
                "transforms": {
                    "timedata": "Hydrogen",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "discharge_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "can_expand": true,
                        "can_retire": false,
                        "has_capacity": true,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true
                        }
                    },
                    "charge_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": true,
                        "can_retire": false,
                        "constraints": {
                            "CapacityConstraint": true
                        }
                    },
                    "compressor_elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "compressor_gas_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": false
                    }
                },
                "storage": {
                    "commodity": "Hydrogen",
                    "can_expand": true,
                    "can_retire": false,
                    "constraints": {
                        "StorageCapacityConstraint": true,
                        "BalanceConstraint": true,
                        "MinStorageLevelConstraint": true
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_Above_ground_storage",
                    "transforms": {
                        "electricity_consumption": 0.018029457
                    },
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "h2_SE",
                            "existing_capacity": 0,
                            "investment_cost": 0.0,
                            "fixed_om_cost": 0.0,
                            "variable_om_cost": 0.0,
                            "efficiency": 1.0,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1
                        },
                        "charge_edge":{
                            "existing_capacity": 0,
                            "investment_cost": 3219.236569,
                            "fixed_om_cost": 0.0,
                            "variable_om_cost": 0.0,
                            "efficiency": 1.0
                        },
                        "compressor_gas_edge": {
                            "start_vertex": "h2_SE"
                        },
                        "compressor_elec_edge": {
                            "start_vertex": "elec_SE"
                        }
                    },
                    "storage":{
                        "investment_cost_storage": 873.013307,
                        "fixed_om_cost_storage":28.75810056,
                        "storage_loss_fraction": 0.0,
                        "min_storage_level": 0.3
                    }
                }
            ]
        }
    ]
}
```

Exercise 2

Following the same steps taken in Tutorial 2, load the input files, generate MACRO model, and solve it using the open-source solver HiGHS.

Solution

First, load the inputs:

```
system = Macro.load_system("one_zone_multisector");
```

Then, generate the model:

```
model = Macro.generate_model(system)
```

Finally, solve it using the HiGHS solver:

```
Macro.set_optimizer(model, HiGHS.Optimizer);
Macro.optimize!(model)
```

Exercise 3

As in Tutorial 2, print optimized capacity for each asset, the system total cost, and the total emissions.

What do you observe?

To explain the results, plot both the electricity generation and hydrogen supply results as done in Tutorial 2 using `VegaLite.jl`.

Solution

Optimized capacities are retrieved as follows:

```
capacity_results = Macro.get_optimal_asset_capacity(system)
```

Total system cost is:

```
Macro.objective_value(model)
```

Total CO_2 emissions are:

```
co2_node = Macro.get_nodes_sametype(system.locations, CO2)[1]
Macro.value(sum(co2_node.operation_expr[:emissions]))
```

Note that we have achieved lower costs and emissions when able to co-optimize capacity and operation of electricity and hydrogen sectors. In the following, we further investigate these

```
plot_time_interval = 3600:3624
```

Here is the electricity generation profile:

```
natgas_power =  Macro.value.(Macro.flow(system.assets[4].elec_edge)).data[plot_time_interval]/1e3;
solar_power = Macro.value.(Macro.flow(system.assets[5].edge)).data[plot_time_interval]/1e3;
wind_power = Macro.value.(Macro.flow(system.assets[6].edge)).data[plot_time_interval]/1e3;

elec_gen =  DataFrame( hours = plot_time_interval,
                solar_photovoltaic = solar_power,
                wind_turbine = wind_power,
                natural_gas_fired_combined_cycle = natgas_power,
                )
```

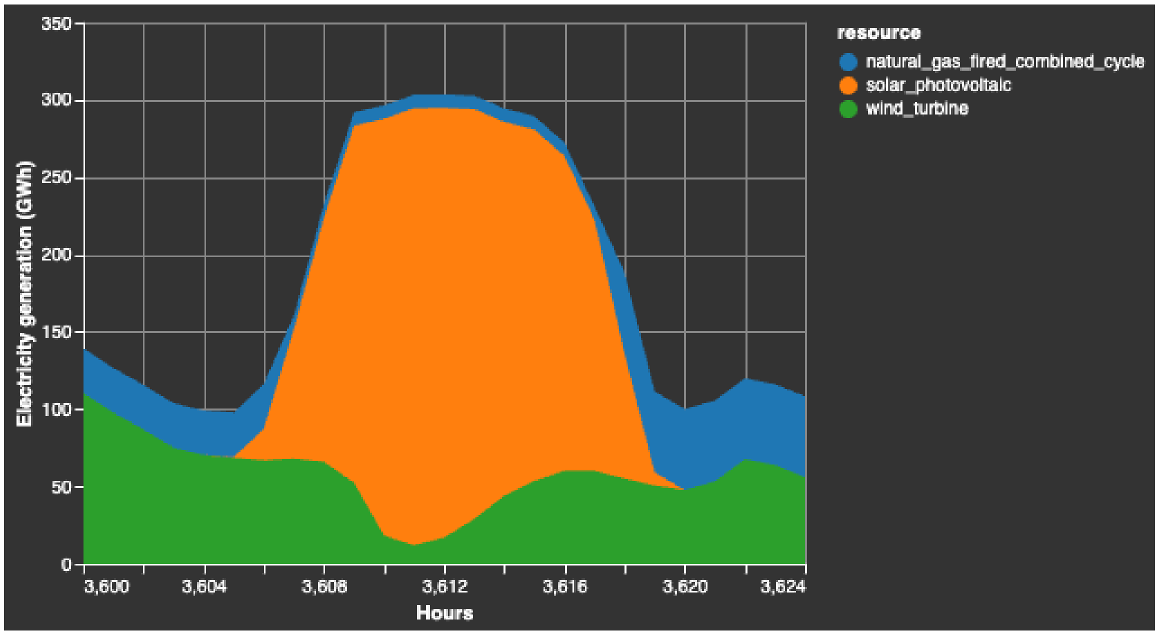

Figure 7.3: elec_generation

```
stack_elec_gen = stack(elec_gen,
↪   [:natural_gas_fired_combined_cycle,:wind_turbine,:solar_photovoltaic], variable_name=:resource,
↪   value_name=:generation);

elc_plot = stack_elec_gen |>
@vlplot(
    :area,
    x={:hours, title="Hours"},
    y={:generation, title="Electricity generation (GWh)",stack=:zero},
    color={"resource:n", scale={scheme=:category10}},
    width=400,
    height=300
)
```

During the day, when solar photovoltaic is available, almost all of the electricity generation comes from VREs.

Because hydrogen storage is cheaper than batteries, we expect the system to use the electricity generated during the day to operate the electrolyzers to meet the hydrogen demand, storing the excess hydrogen to be used when solar photolvoltaics can not generate electricity.

We verify our assumption by making a stacked area plot of the hydrogen supply (hydrogen generation net of the hydrogen stored):

```
electrolyzer_idx = findfirst(isa.(system.assets,Electrolyzer).==1)
h2stor_idx = findfirst(isa.(system.assets,GasStorage{Hydrogen}).==1)

electrolyzer_gen =
↪   Macro.value.(Macro.flow(system.assets[electrolyzer_idx].h2_edge)).data[plot_time_interval]/1e3;
h2stor_charge =
↪   Macro.value.(Macro.flow(system.assets[h2stor_idx].charge_edge)).data[plot_time_interval]/1e3;
```

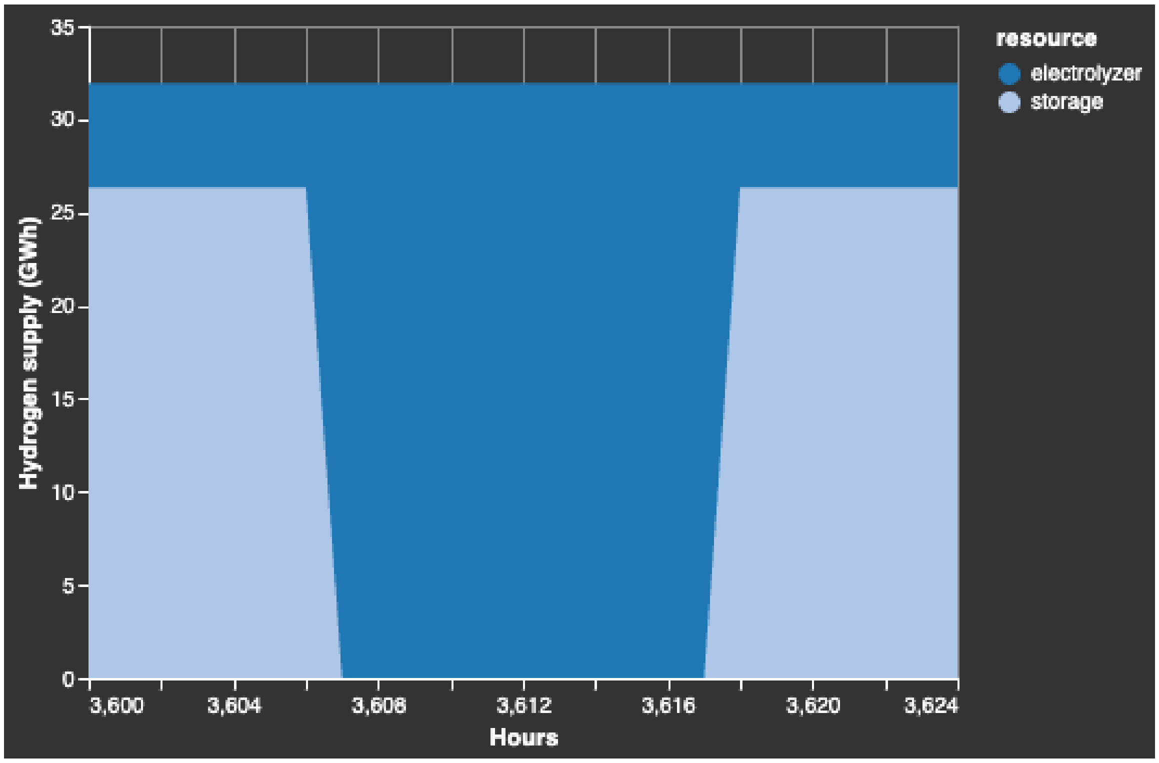

Figure 7.4: h2_generation

```
h2stor_discharge =
↪  Macro.value.(Macro.flow(system.assets[h2stor_idx].discharge_edge)).data[plot_time_interval]/1e3;

h2_gen = DataFrame( hours = plot_time_interval,
                    electrolyzer = electrolyzer_gen - h2stor_charge,
                    storage =  h2stor_discharge)

stack_h2_gen = stack(h2_gen, [:electrolyzer, :storage], variable_name=:resource,
↪  value_name=:supply);

h2plot = stack_h2_gen |>
    @vlplot(
        :area,
        x={:hours, title="Hours"},
        y={:supply, title="Hydrogen supply (GWh)",stack=:zero},
        color={"resource:n", scale={scheme=:category20}},
        width=400,
        height=300
    )
```

Part IV

User Guide

Chapter 8

Sectors

8.1 User Guide

MACRO version 0.1.0

Introduction: Energy System in MACRO

The MACRO model is designed to represent the energy system in a detailed manner, with various sectors and technologies interacting. Each sector is characterized by a **commodity**, a type of energy carrier. The current model includes the following sectors:

- **Electricity**
- **Natural Gas**
- **CO2 and CO2 Capture**
- **Hydrogen**
- **Coal**
- **Biomass**
- **Uranium**

The energy system is modeled as a *multi-plex network* — a multi-layered network that connects different sectors.

The key components of this network are:

1. **Vertices**: Represent **balance equations** and can correspond to transformations (linking two or more commodity networks), storage systems, or demand nodes (outflows):
    - **Transformations**:
        * Special vertices that **convert** one commodity type into another, acting as bridges between sectors.
        * They represent conversion processes defined by a set of **stoichiometric equations** specifying transformation ratios.
    - **Storage**:

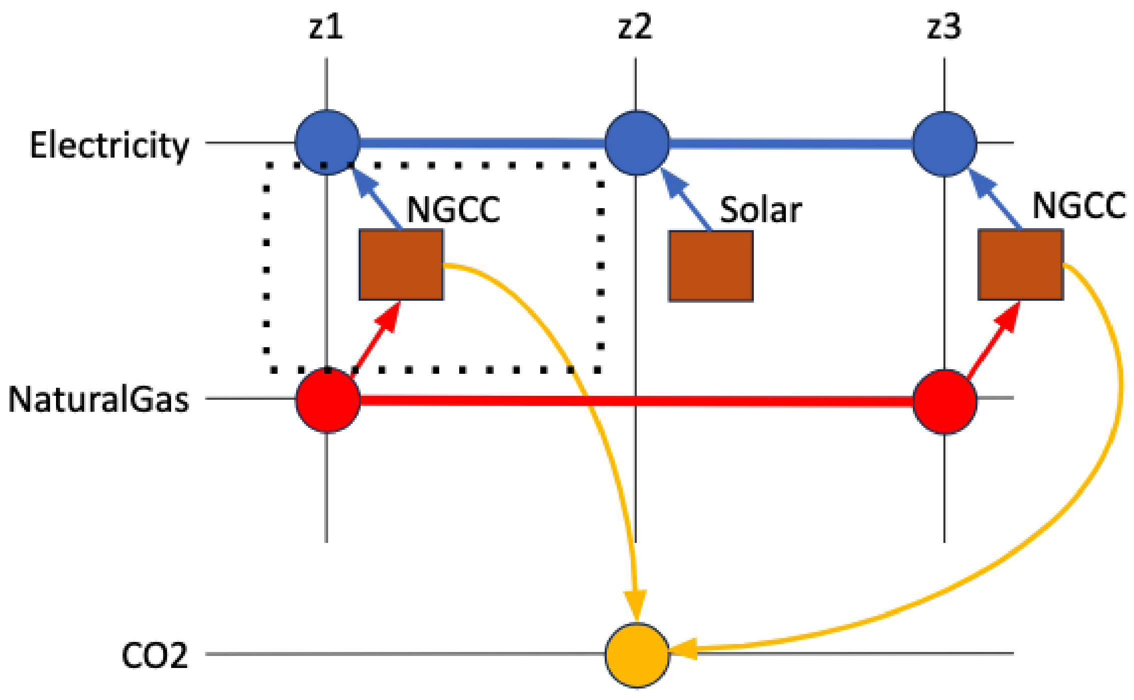

Figure 8.1: Energy System

  - ∗ Stores commodities for future use.
  - ∗ The flow of commodities into and out of storage systems is regulated by **Storage balance** equations.

- **Nodes**:
  - ∗ Represent geographical locations or zones, each associated with a commodity type.
  - ∗ They can be of two types: demand nodes (outflows) or sources (inflows).
  - ∗ **Demand balance** equations are used to balance the flow of commodities into and out of the node.
  - ∗ They form the network for a specific sector (e.g., electricity network, hydrogen network, etc.).

2. **Edges**:
   - Depict the **flow** of a commodity into or out of a vertex.
   - Capacity sizing decisions, capex/opex, planning and operational constraints are associated with the edges.

3. **Assets**: Defined as a collection of edges and vertices. See MACRO Asset Library for a list of all the assets available in MACRO.

The figure below illustrates a multi-plex network representing an energy system with electricity, natural gas, and CO2 sectors, with two natural gas power plants, and a solar panel. Blue nodes represent the electricity sector, red nodes represent natural gas, and yellow nodes represent CO2. The edges depict commodity flow, and squares represent transformation points.

Chapter 9

Input Data

9.1 MACRO Input Data

MACRO version 0.1.0

Tutorial 1

We recommend to check the Tutorial 1 for a step-by-step guide on how to create the input data.

All input files are divided into **three** main folders:

- **settings**: Contains all the settings for the run and the solver.
- **system**: Contains all files related to the system, such as sectors, time resolution, nodes, demand, etc.
- **assets**: Contains all the files that define the assets, such as transmission lines, power plants, storage units, etc.

In the following section, we will go through each folder and file in detail.

Units

Before be dive into the input data, let's define the units of the input data:

Sector/Quantity	Units
Electricity	MWh
Hydrogen	MWh
NaturalGas	MWh
Uranium	MWh
Coal	MWh
CO2	ton
CO2Captured	ton
Biomass	ton
Time	hours
Price	USD

Commodities that require only an energy representation (e.g., Hydrogen) have units of MWh. Commodities that require a physical representation (e.g., Biomass, where regional supply curve is important) have units of metric tonnes. The recommended convention is MWh on a higher heating value basis for transformations where hydrogen is involved, and tonnes on a dry basis for transformations where biomass is involved.

Settings folder

The `settings` folder currently contains only one file, `macro_settings.yml`, which contains the settings for the run.

macro_settings.json

Format: JSON

Attribute	Values	Default	Description
Scaling	True, False	False	If true, the model will scale the input data to the following units: MWh → GWh, tons → ktons, $\/MWh → M$\/GWh, $\/ton → M$\/kton

System folder

The `system` folder currently contains five main files:

- `commodities.json`: Defines the sectors/commodities used in the system.
- `time_data.json`: Defines the time resolution data for each sector.
- `nodes.json`: Defines the nodes in the system.
- `demand.csv`: Contains the demand data.
- `fuel_prices.csv`: Contains the prices of fuels.

commodities.json

Format: JSON

This file contains a list of sectors/commodities used in the system. This is how the file is structured:

```
{
    "commodities": [
        "Sector_1",
        "Sector_2",
        ...
    ]
}
```

For instance, if we want to include the electricity, hydrogen, natural gas, CO2, uranium, and coal sectors, the file should look like this:

```
{
    "commodities": [
        "Electricity",
        "Hydrogen",
        "NaturalGas",
        "CO2",
        "Uranium",
        "Coal"
    ]
}
```

time_data.json

Format: JSON

This file contains the data related to the time resolution for each sector. The file is structured as follows:

```
{
    "PeriodLength": <Integer>,  // units: hours
    "HoursPerTimeStep": {
        "Sector_1": <Integer>,  // units: hours
        "Sector_2": <Integer>,  // units: hours
        ...
    },
    "HoursPerSubperiod": {
        "Sector_1": <Integer>,
        "Sector_2": <Integer>,
        ...
    }
}
```

Attribute	Values	Description
PeriodLength	Integer	Total number of **hours** in the simulation.
HoursPerTimeStep	Integer	Number of **hours** in each time step **for each sector**.
HoursPerSubperiod	Integer	Number of **hours** in each subperiod **for each sector**.

Subperiods

Subperiods represent the time slices of the simulation used to perform time wrapping for time-coupling constraints (see, for example, Macro.timestepbefore).

For instance, if we want to run the model for one year (non leap year), with one hour per time step and a single subperiod, the file should look like this:

```
{
    "PeriodLength": 8760,  // one year
    "HoursPerTimeStep": {
        "Electricity": 1,
        "Hydrogen": 1,
        "NaturalGas": 1,
        "CO2": 1,
        "Uranium": 1,
        "Coal": 1
    },
    "HoursPerSubperiod": {
        "Electricity": 8760,
        "Hydrogen": 8760,
        "NaturalGas": 8760,
        "CO2": 8760,
        "Uranium": 8760,
        "Coal": 8760
    }
}
```

A more complex example is the following:

```
{
    "PeriodLength": 504,  // 3 weeks
    "HoursPerTimeStep": {
        "Electricity": 1,
        "Hydrogen": 1,
        "NaturalGas": 1,
        "CO2": 1,
        "Uranium": 1,
        "Coal": 1
    },
    "HoursPerSubperiod": {
        "Electricity": 168,  // 1 week
        "Hydrogen": 168,
        "NaturalGas": 168,
        "CO2": 168,
        "Uranium": 168,
        "Coal": 168
    }
}
```

In this example, the simulation will run for 504 hours (3 weeks), with one hour per time step and 1 week per subperiod.

nodes.json

Format: JSON

This file defines the regions/nodes for each sector. It is structured as a list of dictionaries, where each dictionary defines a network for a given sector.

Each dictionary (network) has three main attributes:

- `type`: The type of the network (e.g. "NaturalGas", "Electricity", etc.).
- `global_data`: attributes that are the same for all the nodes in the network.
- `instance_data`: attributes that are different for each node in the network.

This structure for the network has the advantage of **grouping the common attributes** for all the nodes in a single dictionary, avoiding to repeat the same attribute for each node.

Node attributes

The `Node` object is defined in the file `nodes.jl` and can be found here Macro.Node.

Here is an example of a `nodes.json` file with both electricity, natural gas, CO2 and biomass sectors covering most of the attributes present above. The (multiplex)-network in the system is made of the following networks:

- NaturalGas
  - `natgas_SE`
  - `natgas_MIDAT`

Attribute	Type	Values	Default	Description
id	`String`	`String`	Required	Unique identifier for the node. E.g. "elec_node_1".
type	`String`	Any MACRO commodity type	Required	Commodity type. E.g. "Electricity".
time_interval	`String`	Any MACRO commodity type	Required	Time resolution for the time series data linked to the node. E.g. "Electricity".
constraints	`Dict{String,Bool}`	Any MACRO constraint type	Empty	List of constraints applied to the node. E.g. `{"BalanceConstraint": true, "MaxNonServedDemandConstraint": true}`.
demand	`Dict`	Demand file path and header	Empty	Path to the demand file and column name for the demand time series to link to the node. E.g. `{"timeseries": {"path": "system/demand.csv", "header": "Demand_MW_z1"}}`.
price	`Dict`	Price file path and header	Empty	Path to the price file and column name for the price time series to link to the node. E.g. `{"timeseries": {"path": "system/fuel_prices.csv", "header": "natgas_SE"}}`.
max_nsd	`Vector{Float64}`	Vector of numbers ∈ [0,1]	[0.0]	Maximum allowed non-served demand for each demand segment as a fraction of the total demand. E.g. `[1.0]` for a single segment.
price_nsd	`Vector{Float64}`	Vector of numbers	[0.0]	Price/penalty for non-served demand by segment. E.g. `[5000.0]` for a single segment.
price_supply	`Vector{Float64}`	Vector of numbers	[0.0]	Piecewise linear price for supply curves. E.g. `[0.0, 100.0, 200.0]`.
max_supply	`Vector{Float64}`	Vector of numbers	[0.0]	Maximum allowed supply for each supply segment. E.g. `[1000.0]` for a single segment.
rhs_policy	`Dict{DataType,Float64}`	Dict of MACRO constraint types and numbers	Empty	Right hand side of the policy constraints. E.g. `{"CO2CapConstraint": 200}`, carbon price of 200 USD/ton.
price_unmet_policy	`Dict{DataType,Float64}`	Dict of MACRO policy types and numbers	Empty	Price/penalty for unmet policy constraints.

  - `natgas_NE`
- Electricity
  - `elec_SE`
  - `elec_MIDAT`
  - `elec_NE`
- CO2
  - `co2_sink`
- Biomass

- `bioherb_SE`

Therefore, the system has 4 networks and 8 nodes in total.

```
{
    "nodes": [
        {
            "type": "NaturalGas",
            "global_data": {
                "time_interval": "NaturalGas" // time resolution as defined in the time_data.json
                ↪  file
            },
            "instance_data": [
                {
                    "id": "natgas_SE",
                    "price": {
                        "timeseries": {
                            "path": "system/fuel_prices.csv", // path to the price file
                            "header": "natgas_SE" // column name in the price file for the price
                            ↪  time series
                        }
                    }
                },
                {
                    "id": "natgas_MIDAT",
                    "price": {
                        "timeseries": {
                            "path": "system/fuel_prices.csv",
                            "header": "natgas_MIDAT"
                        }
                    }
                },
                {
                    "id": "natgas_NE",
                    "price": {
                        "timeseries": {
                            "path": "system/fuel_prices.csv",
                            "header": "natgas_NE"
                        }
                    }
                }
            ]
        },
        {
            "type": "Electricity",
            "global_data": {
                "time_interval": "Electricity",
                "max_nsd": [
                    1
                ],
                "price_nsd": [
                    5000.0
                ],
                "constraints": {    // constraints applied to the nodes
                    "BalanceConstraint": true,
                    "MaxNonServedDemandConstraint": true,
                    "MaxNonServedDemandPerSegmentConstraint": true
                }
            },
            "instance_data": [
                {
                    "id": "elec_SE",
                    "demand": {
                        "timeseries": {
                            "path": "system/demand.csv", // path to the demand file
                            "header": "Demand_MW_z1" // column name in the demand file for the
                            ↪  demand time series
                        }
                    }
                },
                {
                    "id": "elec_MIDAT",
                    "demand": {
                        "timeseries": {
                            "path": "system/demand.csv",
                            "header": "Demand_MW_z2"
                        }
                    }
                },
                {
                    "id": "elec_NE",
                    "demand": {
                        "timeseries": {
                            "path": "system/demand.csv",
                            "header": "Demand_MW_z3"
                        }
                    }
                }
            ]
        },
        {
            "type": "CO2",
            "global_data": {
                "time_interval": "CO2"
            },
            "instance_data": [
                {
                    "id": "co2_sink",
                    "constraints": {
                        "CO2CapConstraint": true
                    },
                    "rhs_policy": {
                        "CO2CapConstraint": 0
                    },
                    "price_unmet_policy": {
                        "CO2CapConstraint": 250.0
                    }
                }
            ]
        },
        {
            "type": "Biomass",
            "global_data": {
                "time_interval": "Biomass",
                "constraints": {
                    "BalanceConstraint": true
                }
            },
            "instance_data": [
                {
                    "id": "bioherb_SE",
                    "demand": {
                        "timeseries": {
                            "path": "system/demand.csv",
                            "header": "Demand_Zero"
                        }
                    },
                    "max_supply": [
                        10000,
                        20000,
                        30000
                    ],
                    "price_supply": [
                        40,
                        60,
                        80
                    ]
                }
            ]
        }
    ]
}
```

demand.csv

Format: CSV

This file contains the demand data for each region/node.

- First column: Time step.
- Remaining columns: Demand for each region/node (units: MWh).

Example:

TimeStep	DemandMWz1	DemandMWz2	DemandMWz3
1	100	200	300
2	110	210	310
...	...	...	...

fuel_prices.csv

Format: CSV

This file contains the prices for each fuel for each region/node.

- First column: Time step.
- Remaining columns: Prices for each region/node (units: USD/MWh).

Example:

TimeStep	natgas_SE	natgas_MIDAT	natgas_NE
1	100	110	120
2	110	120	130
...	...	...	...

Assets folder

The `assets` folder contains all the files that define the resources and technologies that are included in the system. As a general rule, each asset type has its own file, where each file is structured in a similar way to the `nodes.json` file.

Asset type files

Format: JSON

Each asset type file has the following three main parameters:

- `type`: The type of the asset (e.g. "Battery", "FuelCell", "PowerLine", etc.).
- `global_data`: attributes that are the same for all the assets of the same type.
- `instance_data`: attributes that are different for each asset of the same type.

Depending on the graph structure of the asset, both `global_data` and `instance_data` can have different attributes, one for each transformation, edge, and storage present in the asset.

Example: natural gas power plant

For example, a natural gas combined cycle power plant is represented by an asset made of:

- **1 transformation** (combustion and electricity generation)
- **3 edges**
  - natural gas flow
  - electricity flow
  - CO2 flow

Then, both `global_data` and `instance_data` will have the following structure:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "fuel_edge": {
            // ... fuel_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        }
    }
}
```

In the following sections, we will go through each asset type and show the attributes that can be set for each of them. Each section will contain the following three parts:

- **Graph structure**: a graphical representation of the asset, showing the transformations, edges, and storages present in the asset.
- **Attributes**: a table with the attributes that can be set for each asset type.
- **Example**: an example of the asset type file (`.json`).

9.2 Example of the folder structure for the input data

```
MacroCase
│
├── 📁 settings
│   └── macro_settings.yml
│
├── 📁 system
│   ├── commodities.json
│   ├── time_data.json
│   ├── nodes.json
│   ├── demand.csv
│   └── fuel_prices.csv
│
├── ▯ assets
│   ├──battery.json
│   ├──electrolyzers.json
│   ├──fuel_prices.csv
│   ├──fuelcell.json
│   ├──h2storage.json
│   ├──power_lines.json
│   ├──thermal_h2.json
│   ├──thermal_power.json
│   ├──vre.json
│ [...other asset types...]
│   └──availability.csv
│
└── system_data.json
```

Chapter 10

Assets

10.1 MACRO Asset Library

MACRO is designed to be a flexible and modular model that can adapt to various energy system representations. The model includes a library of assets that represent different technologies within the energy system.

Each asset is defined by a **combination of transformations, edges, and storage units** that represent the physical and operational characteristics of a technology. These assets can be combined to create a detailed representation of the energy system, capturing the interactions between technologies and sectors.

In the following sections, we will introduce each asset type and show the **attributes** that can be set for each of them as well as the **equations** that define the conversion processes. We will also provide a **graphical representation** of the asset in terms of transformations, edges, and storages to help the user understand the structure of the asset.

Each section will have the following three parts:

1. **Graph structure**: a graphical representation of the asset, showing the transformations, edges, and storages present in the asset.
2. **Attributes**: a table with the attributes that can be set for each asset type.
3. **Example**: an example of the asset type input file (`.json`) that can be used to create the asset.

Available assets

The current library includes the following assets:

- Battery
- BECCS Electricity
- BECCS Hydrogen
- Electric DAC
- Electrolyzer
- Fuel Cell

- Gas Storage
- Hydrogen Line
- Hydro Reservoir
- Must Run
- Natural Gas DAC
- Power Line
- Thermal Hydrogen Plant (with and without CCS)
- Thermal Power Plant (with and without CCS)
- Variable Renewable Energy resources (VRE)

10.2 Battery

Graph structure

A battery is a storage technology that is represented in MACRO by the following graph structure:

Therefore, a battery asset is made of:

- 1 `Storage` component, representing the battery storage.
- 2 `Electricity` Edge components:
  - one **incoming** representing the charge edge from the electricity network to the storage.
  - one **outgoing** representing the discharge edge from the storage to the electricity network.

Attributes

As for all the other assets, the structure of the input file for a battery asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "storage":{
        // ... storage-specific attributes ...
    },
    "edges":{
        "charge_edge": {
            // ... charge_edge-specific attributes ...
        },
        "discharge_edge": {
            // ... discharge_edge-specific attributes ...
        }
    }
}
```

where the possible attributes that the user can set are reported in the following tables.

Storage component

The definition of the `Storage` object can be found here Macro.Storage.

Default constraints

As noted in the above table, the **default constraints** for the storage component of the battery are the following:

- Balance constraint
- Storage capacity constraint
- Storage max duration constraint
- Storage min duration constraint
- Storage symmetric capacity constraint

Charge and discharge edges

Both the charge and discharge edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Efficiency

The efficiency of the charging/discharging process can be set in the `charge_edge` and `discharge_edge` parts of the input file. These parameters are used, for example, in the Balance constraint to balance the charge and discharge flows.

Default constraints

The **default constraints** for the discharge edge are the following:

- Capacity constraint
- Storage discharge limit constraint
- Ramping limits constraint

Example

The following is an example of the input file for a battery asset that creates three batteries, one in each of the SE, MIDAT and NE regions.

```
{
    "elec_stor": [
        {
            "type": "Battery",
            "global_data": {
                "storage": {
                    "commodity": "Electricity",
                    "can_expand": true,
                    "can_retire": false,
                    "constraints": {
                        "StorageCapacityConstraint": true,
                        "StorageSymmetricCapacityConstraint": true,
                        "StorageMinDurationConstraint": true,
                        "StorageMaxDurationConstraint": true,
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "discharge_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": true,
                        "can_retire": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "StorageDischargeLimitConstraint": true
                        }
                    },
                    "charge_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    }
                }
            },
            "instance_data": [
                {
                    "id": "battery_SE",
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "elec_SE",
                            "capacity_size": 1.0,
                            "existing_capacity": 0.0,
                            "fixed_om_cost": 4536.98,
                            "investment_cost": 17239.56121,
                            "variable_om_cost": 0.15,
                            "efficiency": 0.92
                        },
                        "charge_edge": {
                            "start_vertex": "elec_SE",
                            "efficiency": 0.92,
                            "variable_om_cost": 0.15
                        }
                    },
                    "storage": {
                        "existing_capacity_storage": 0.0,
                        "fixed_om_cost_storage": 2541.19,
                        "investment_cost_storage": 9656.002735,
                        "max_duration": 10,
                        "min_duration": 1
                    }
                },
                {
                    "id": "battery_MIDAT",
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "elec_SE",
                            "capacity_size": 1.0,
                            "existing_capacity": 0.0,
                            "fixed_om_cost": 4536.98,
                            "investment_cost": 17239.56121,
                            "variable_om_cost": 0.15,
                            "efficiency": 0.92
                        },
                        "charge_edge": {
                            "start_vertex": "elec_SE",
                            "efficiency": 0.92,
                            "variable_om_cost": 0.15
                        }
                    },
                    "storage": {
                        "existing_capacity_storage": 0.0,
                        "fixed_om_cost_storage": 2541.19,
                        "investment_cost_storage": 9656.002735,
                        "max_duration": 10,
                        "min_duration": 1
                    }
                },
                {
                    "id": "battery_NE",
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "elec_SE",
                            "capacity_size": 1.0,
                            "existing_capacity": 0.0,
                            "fixed_om_cost": 4536.98,
                            "investment_cost": 17239.56121,
                            "variable_om_cost": 0.15,
                            "efficiency": 0.92
                        },
                        "charge_edge": {
                            "start_vertex": "elec_SE",
                            "efficiency": 0.92,
                            "variable_om_cost": 0.15
                        }
                    },
                    "storage": {
                        "existing_capacity_storage": 0.0,
                        "fixed_om_cost_storage": 2541.19,
                        "investment_cost_storage": 9656.002735,
                        "max_duration": 10,
                        "min_duration": 1
                    }
                }
            ]
        }
    ]
}
```

10.3 BECCS Electricity

Graph structure

Bioenergy with carbon capture and storage (BECCS) that produces electricity is represented in MACRO using the following graph structure:

A BECCS electricity asset is made of:

- 1 `Transformation` component, representing the BECCS process.
- 5 Edge components:
  - 1 **incoming** `Biomass` Edge, representing the biomass supply.
  - 1 **incoming** `CO2` Edge, representing the CO2 that is absorbed by the biomass.
  - 1 **outgoing** `Electricity` Edge, representing the electricity production.
  - 1 **outgoing** `CO2Captured` Edge, representing the CO2 that is captured.
  - 1 **outgoing** `CO2` Edge, representing the CO2 that is emitted.

Attributes

The structure of the input file for a BECCS electricity asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "biomass_edge": {
            // ... biomass_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        },
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "co2_emission_edge": {
            // ... co2_emission_edge-specific attributes ...
        },
        "co2_captured_edge": {
            // ... co2_captured_edge-specific attributes ...
        }
    }
}
```

where the possible attributes that the user can set are reported in the following tables.

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the BECCS electricity asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

BECCSElectricity

$$\begin{aligned}
\phi_{elec} &= \phi_{biomass} \cdot \epsilon_{elec\_prod} \\
\phi_{co2} &= -\phi_{biomass} \cdot \epsilon_{co2} \\
\phi_{co2} &= \phi_{biomass} \cdot \epsilon_{emission\_rate} \\
\phi_{co2\_captured} &= \phi_{biomass} \cdot \epsilon_{co2\_capture\_rate}
\end{aligned}$$

Edges

Asset expansion

As a modeling decision, only the `Biomass` edge is allowed to expand. Consequently, the `has_capacity` and `constraints` attributes can only be set for the `Biomass` edge. For all other edges, these attributes are pre-set to false and an empty list, respectively, to ensure proper modeling of the asset.

Directionality

The `unidirectional` attribute is only available for the `Biomass` edge. For the other edges, this attribute is pre-set to `true` to ensure the correct modeling of the asset.

All the edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Default constraints

The only **default constraint** for the edges of the BECCS electricity asset is the Capacity constraint applied to the `Biomass` edge.

Example

The following is an example of the input file for a BECCS electricity asset that creates six BECCS electricity assets, two in each of the SE, MIDAT and NE regions.

```
{
    "BECCS_Electricity": [
        {
            "type": "BECCSElectricity",
            "global_data": {
                "transforms": {
                    "timedata": "Biomass",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "biomass_edge": {
                        "type": "Biomass",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": true,
                        "can_retire": true,
                        "integer_decisions": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "MinFlowConstraint": true
                        }
                    },
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "start_vertex": "co2_sink"
                    },
                    "co2_emission_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_sink"
                    },
                    "co2_captured_edge": {
                        "type": "CO2Captured",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_captured_sink"
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_BECCS_Electricity_Herb",
                    "transforms": {
                        "electricity_production": 1.656626506,
                        "capture_rate": 1.5313914,
                        "co2_content": 1.76022,
                        "emission_rate": 0.2288286
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "bioherb_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_BECCS_Electricity_Herb"
                                }
                            },
                            "investment_cost": 696050.2868,
                            "fixed_om_cost": 193228.9157,
                            "variable_om_cost": 42.93975904,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.4
                        },
                        "elec_edge": {
                            "end_vertex": "elec_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_BECCS_Electricity_Herb",
                    "transforms": {
                        "electricity_production": 1.656626506,
                        "capture_rate": 1.5313914,
                        "co2_content": 1.76022,
                        "emission_rate": 0.2288286
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "bioherb_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_BECCS_Electricity_Herb"
                                }
                            },
                            "investment_cost": 696050.2868,
                            "fixed_om_cost": 193228.9157,
                            "variable_om_cost": 42.93975904,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.4
                        },
                        "elec_edge": {
                            "end_vertex": "elec_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_BECCS_Electricity_Herb",
                    "transforms": {
                        "electricity_production": 1.656626506,
                        "capture_rate": 1.5313914,
                        "co2_content": 1.76022,
                        "emission_rate": 0.2288286
                    },
                    "edges": {
                        "biomass_edge": {
```

```
                "start_vertex": "bioherb_NE",
                "availability": {
                    "timeseries": {
                        "path": "assets/availability.csv",
                        "header": "NE_BECCS_Electricity_Herb"
                    }
                },
                "investment_cost": 696050.2868,
                "fixed_om_cost": 193228.9157,
                "variable_om_cost": 42.93975904,
                "capacity_size": 400,
                "min_flow_fraction": 0.4
            },
            "elec_edge": {
                "end_vertex": "elec_NE"
            }
        }
    },
    {
        "id": "SE_BECCS_Electricity_Wood",
        "transforms": {
            "electricity_production": 1.656626506,
            "capture_rate": 1.5313914,
            "co2_content": 1.76022,
            "emission_rate": 0.2288286
        },
        "edges": {
            "biomass_edge": {
                "start_vertex": "biowood_SE",
                "availability": {
                    "timeseries": {
                        "path": "assets/availability.csv",
                        "header": "SE_BECCS_Electricity_Wood"
                    }
                },
                "investment_cost": 696050.2868,
                "fixed_om_cost": 193228.9157,
                "variable_om_cost": 42.93975904,
                "capacity_size": 400,
                "min_flow_fraction": 0.4
            },
            "elec_edge": {
                "end_vertex": "elec_SE"
            }
        }
    },
    {
        "id": "MIDAT_BECCS_Electricity_Wood",
        "transforms": {
            "electricity_production": 1.656626506,
            "capture_rate": 1.5313914,
            "co2_content": 1.76022,
            "emission_rate": 0.2288286
        },
        "edges": {
```

```
                        "biomass_edge": {
                            "start_vertex": "biowood_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_BECCS_Electricity_Wood"
                                }
                            },
                            "investment_cost": 696050.2868,
                            "fixed_om_cost": 193228.9157,
                            "variable_om_cost": 42.93975904,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.4
                        },
                        "elec_edge": {
                            "end_vertex": "elec_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_BECCS_Electricity_Wood",
                    "transforms": {
                        "electricity_production": 1.656626506,
                        "capture_rate": 1.5313914,
                        "co2_content": 1.76022,
                        "emission_rate": 0.2288286
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "biowood_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_BECCS_Electricity_Wood"
                                }
                            },
                            "investment_cost": 696050.2868,
                            "fixed_om_cost": 193228.9157,
                            "variable_om_cost": 42.93975904,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.4
                        },
                        "elec_edge": {
                            "end_vertex": "elec_NE"
                        }
                    }
                }
            ]
        }
    ]
}
```

10.4 BECCS Hydrogen

Graph structure

Bioenergy with carbon capture and storage (BECCS) that produces hydrogen is represented in MACRO using the following graph structure:

A BECCS hydrogen asset is made of:

- 1 `Transformation` component, representing the BECCS process.
- 6 Edge components:
  - 1 **incoming** `Biomass` Edge, representing the biomass supply.
  - 1 **incoming** `CO2` Edge, representing the CO2 that is absorbed by the biomass.
  - 1 **incoming** `Electricity` Edge, representing the electricity consumption.
  - 1 **outgoing** `Hydrogen` Edge, representing the hydrogen production.
  - 1 **outgoing** `CO2Captured` Edge, representing the CO2 that is captured.
  - 1 **outgoing** `CO2` Edge, representing the CO2 that is emitted.

Attributes

The structure of the input file for a BECCS hydrogen asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "biomass_edge": {
            // ... biomass_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        },
        "h2_edge": {
            // ... h2_edge-specific attributes ...
        },
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "co2_emission_edge": {
            // ... co2_emission_edge-specific attributes ...
        },
        "co2_captured_edge": {
            // ... co2_captured_edge-specific attributes ...
        }
    }
}
```

where the possible attributes that the user can set are reported in the following tables.

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the BECCS hydrogen asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

BECCSHydrogen

$$\begin{aligned}
\phi_{h2} &= \phi_{biomass} \cdot \epsilon_{h2\_prod} \\
\phi_{elec} &= -\phi_{biomass} \cdot \epsilon_{elec\_consumption} \\
\phi_{co2} &= -\phi_{biomass} \cdot \epsilon_{co2} \\
\phi_{co2} &= \phi_{biomass} \cdot \epsilon_{emission\_rate} \\
\phi_{co2\_captured} &= \phi_{biomass} \cdot \epsilon_{co2\_capture\_rate}
\end{aligned}$$

Edges

Asset expansion

As a modeling decision, only the `Biomass` edge is allowed to expand. Consequently, the `has_capacity` and `constraints` attributes can only be set for the `Biomass` edge. For all other edges, these attributes are pre-set to false and an empty list, respectively, to ensure proper modeling of the asset.

Directionality

The `unidirectional` attribute is only available for the `Biomass` edge. For the other edges, this attribute is pre-set to `true` to ensure the correct modeling of the asset.

All the edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Default constraints

The only **default constraint** for the edges of the BECCS hydrogen asset is the Capacity constraint applied to the `Biomass` edge.

Example

The following is an example of the input file for a BECCS hydrogen asset that creates six BECCS hydrogen assets, two in each of the SE, MIDAT and NE regions.

```json
{
    "BECCS_Hydrogen": [
        {
            "type": "BECCSHydrogen",
            "global_data": {
                "transforms": {
                    "timedata": "Biomass",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "h2_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "biomass_edge": {
                        "type": "Biomass",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": true,
                        "can_retire": true,
                        "integer_decisions": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "MinFlowConstraint": true
                        }
                    },
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "start_vertex": "co2_sink"
                    },
                    "co2_emission_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_sink"
                    },
                    "co2_captured_edge": {
                        "type": "CO2Captured",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_captured_sink"
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_BECCS_H2_Herb",
                    "transforms": {
                        "hydrogen_production": 2.614520322,
                        "electricity_consumption": 0.083426966,
                        "capture_rate": 1.50022,
                        "co2_content": 1.76022,
                        "emission_rate": 0.26
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "bioherb_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_BECCS_H2_Herb"
                                }
                            },
                            "investment_cost": 532452.9904,
                            "fixed_om_cost": 60067.41573,
                            "variable_om_cost": 38.44314607,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.85
                        },
                        "elec_edge": {
                            "start_vertex": "elec_SE"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_BECCS_H2_Herb",
                    "transforms": {
                        "hydrogen_production": 2.614520322,
                        "electricity_consumption": 0.083426966,
                        "capture_rate": 1.50022,
                        "co2_content": 1.76022,
                        "emission_rate": 0.26
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "bioherb_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_BECCS_H2_Herb"
                                }
                            },
                            "investment_cost": 532452.9904,
                            "fixed_om_cost": 60067.41573,
                            "variable_om_cost": 38.44314607,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.85
                        },
                        "elec_edge": {
                            "start_vertex": "elec_MIDAT"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_BECCS_H2_Herb",
                    "transforms": {
                        "hydrogen_production": 2.614520322,
                        "electricity_consumption": 0.083426966,
                        "capture_rate": 1.50022,
                        "co2_content": 1.76022,
                        "emission_rate": 0.26
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "bioherb_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_BECCS_H2_Herb"
                                }
                            },
                            "investment_cost": 532452.9904,
                            "fixed_om_cost": 60067.41573,
                            "variable_om_cost": 38.44314607,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.85
                        },
                        "elec_edge": {
                            "start_vertex": "elec_NE"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_NE"
                        }
                    }
                },
                {
                    "id": "SE_BECCS_H2_Wood",
                    "transforms": {
                        "hydrogen_production": 2.614520322,
                        "electricity_consumption": 0.083426966,
                        "capture_rate": 1.50022,
                        "co2_content": 1.76022,
                        "emission_rate": 0.26
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "biowood_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_BECCS_H2_Wood"
                                }
                            },
                            "investment_cost": 532452.9904,
                            "fixed_om_cost": 60067.41573,
                            "variable_om_cost": 38.44314607,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.85
                        },
                        "elec_edge": {
                            "start_vertex": "elec_SE"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_BECCS_H2_Wood",
                    "transforms": {
                        "hydrogen_production": 2.614520322,
                        "electricity_consumption": 0.083426966,
                        "capture_rate": 1.50022,
                        "co2_content": 1.76022,
                        "emission_rate": 0.26
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "biowood_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_BECCS_H2_Wood"
                                }
                            },
                            "investment_cost": 532452.9904,
                            "fixed_om_cost": 60067.41573,
                            "variable_om_cost": 38.44314607,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.85
                        },
                        "elec_edge": {
                            "start_vertex": "elec_MIDAT"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_BECCS_H2_Wood",
                    "transforms": {
                        "hydrogen_production": 2.614520322,
                        "electricity_consumption": 0.083426966,
                        "capture_rate": 1.50022,
                        "co2_content": 1.76022,
                        "emission_rate": 0.26
                    },
                    "edges": {
                        "biomass_edge": {
                            "start_vertex": "biowood_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_BECCS_H2_Wood"
                                }
                            },
                            "investment_cost": 532452.9904,
                            "fixed_om_cost": 60067.41573,
                            "variable_om_cost": 38.44314607,
                            "capacity_size": 400,
                            "min_flow_fraction": 0.85
                        },
                        "elec_edge": {
                            "start_vertex": "elec_NE"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_NE"
                        }
                    }
                }
            ]
        }
    ]
}
```

10.5 Electric DAC

Graph structure

An electric direct air capture (DAC) asset is represented in MACRO using the following graph structure:

An electric DAC asset is made of:

- 1 `Transformation` component, representing the DAC process.
- 3 Edge components:
  - 1 **incoming** `Electricity` Edge, representing the electricity consumption.
  - 1 **incoming** `CO2` Edge, representing the CO2 that is captured.
  - 1 **outgoing** `CO2 Captured` Edge, representing the CO2 that is captured.

Attributes

The structure of the input file for an electric DAC asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        },
        "co2_captured_edge": {
            // ... co2_captured_edge-specific attributes ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the ElectricDAC asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

ElectricDAC

$$\phi_{elec} = \phi_{co2\_captured} \cdot \epsilon_{elec\_consumption}$$
$$\phi_{co2} = \phi_{co2\_captured}$$

Edge

Both the incoming and outgoing edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Default constraints

The only **default constraint** for the edges of the ElectricDAC asset is the Capacity constraint applied to the CO2 edge.

Example

The following is an example of the input file for an ElectricDAC asset that creates three electric DAC units, each for a different region.

```
{
    "ElectricDAC": [
        {
            "type": "ElectricDAC",
            "global_data": {
                "transforms": {
                    "timedata": "Electricity",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": true,
                        "start_vertex": "co2_sink",
                        "can_retire": true,
                        "can_expand": true,
                        "uc": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true
                        },
                        "integer_decisions": false
                    },
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "co2_captured_edge": {
                        "type": "CO2Captured",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_captured_sink"
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_Solvent_DAC",
                    "transforms": {
                        "electricity_consumption": 4.38
                    },
                    "edges": {
                        "co2_edge": {
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_Solvent_DAC"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 939000.00,
                            "fixed_om_cost": 747000.00,
                            "variable_om_cost": 22.00,
                            "ramp_up_fraction": 1.0,
                            "ramp_down_fraction": 1.0
                        },
                        "elec_edge": {
                            "start_vertex": "elec_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_Solvent_DAC",
                    "transforms": {
                        "electricity_consumption": 4.38
                    },
                    "edges": {
                        "co2_edge": {
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_Solvent_DAC"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 939000.00,
                            "fixed_om_cost": 747000.00,
                            "variable_om_cost": 22.00,
                            "ramp_up_fraction": 1.0,
                            "ramp_down_fraction": 1.0
                        },
                        "elec_edge": {
                            "start_vertex": "elec_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_Solvent_DAC",
                    "transforms": {
                        "electricity_consumption": 4.38
                    },
                    "edges": {
                        "co2_edge": {
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_Solvent_DAC"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 939000.00,
                            "fixed_om_cost": 747000.00,
                            "variable_om_cost": 22.00,
                            "ramp_up_fraction": 1.0,
                            "ramp_down_fraction": 1.0
                        },
                        "elec_edge": {
                            "start_vertex": "elec_NE"
                        }
                    }
                }
            ]
        }
    ]
}
```

10.6 Electrolyzer

Graph structure

An electrolyzer asset is represented in MACRO using the following graph structure:

An electrolyzer asset is made of:

- 1 `Transformation` component, representing the electrolysis process.
- 2 Edge components:
    - 1 **incoming** `Electricity` Edge, representing the electricity consumption.
    - 1 **outgoing** `Hydrogen` Edge, representing the hydrogen production.

Attributes

The structure of the input file for an electrolyzer asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "h2_edge": {
            // ... co2_edge-specific attributes ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the electrolyzer asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

Electrolyzer

$$\phi_{h2} = \phi_{elec} \cdot \epsilon_{efficiency}$$

Edges

Both the electricity and hydrogen edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Default constraints

The only **default constraint** for the edges of the electrolyzer asset is the Capacity constraint applied to the hydrogen edge.

Example

The following is an example of the input file for an electrolyzer asset that creates three electrolyzers, each for each of the SE, MIDAT and NE regions.

```
{
    "electrolyzer": [
        {
            "type": "Electrolyzer",
            "global_data": {
                "nodes": {},
                "transforms": {
                    "timedata": "Electricity",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "h2_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_retire": true,
                        "can_expand": true,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true,
                            "MinFlowConstraint": true
                        }
                    },
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_Electrolyzer",
                    "transforms": {
                        "efficiency_rate": 0.875111139
                    },
                    "edges": {
                        "elec_edge": {
                            "start_vertex": "elec_SE"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_SE",
                            "existing_capacity": 0,
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.1
                        }
                    }
                },
                {
                    "id": "MIDAT_Electrolyzer",
                    "transforms": {
                        "efficiency_rate": 0.875111139
                    },
                    "edges": {
                        "elec_edge": {
                            "start_vertex": "elec_MIDAT"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_MIDAT",
                            "existing_capacity": 0,
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.1
                        }
                    }
                },
                {
                    "id": "NE_Electrolyzer",
                    "transforms": {
                        "efficiency_rate": 0.875111139
                    },
                    "edges": {
                        "elec_edge": {
                            "start_vertex": "elec_NE"
                        },
                        "h2_edge": {
                            "end_vertex": "h2_NE",
                            "existing_capacity": 0,
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.1
                        }
                    }
                }
            ]
        }
    ]
}
```

10.7 Fuel Cell

Graph structure

A fuel cell is represented in MACRO using the following graph structure:

A fuel cell asset is made of:

- 1 `Transformation` component, representing the fuel cell process.
- 2 Edge components:
  - 1 **incoming** `Hydrogen` Edge, representing the hydrogen supply.
  - 1 **outgoing** `Electricity` Edge, representing the electricity production.

Attributes

The structure of the input file for a fuel cell asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "h2_edge": {
            // ... h2_edge-specific attributes ...
        },
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the fuel cell asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

Fuel Cell

$$\phi_{elec} = \phi_{h2} \cdot \epsilon_{efficiency}$$

Edges

Both the electricity and hydrogen edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Default constraints

The only **default constraint** for the edges of the fuel cell asset is the Capacity constraint applied to the electricity edge.

Example

The following is an example of the input file for a fuel cell asset that creates three fuel cells, each for each of the SE, MIDAT and NE regions.

```
{
    "fuelcell": [
        {
            "type": "FuelCell",
            "global_data": {
                "nodes": {},
                "transforms": {
                    "timedata": "Hydrogen",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_retire": true,
                        "can_expand": true,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true,
                            "MinFlowConstraint": true
                        }
                    },
                    "h2_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": false
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_Electrolyzer",
                    "transforms": {
                        "efficiency_rate": 0.875111139
                    },
                    "edges": {
                        "h2_edge": {
                            "start_vertex": "h2_SE"
                        },
                        "elec_edge": {
                            "end_vertex": "elec_SE",
                            "existing_capacity": 0,
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.1
                        }
                    }
                },
                {
                    "id": "MIDAT_FuelCell",
                    "transforms": {
                        "efficiency_rate": 0.875111139
                    },
                    "edges": {
                        "h2_edge": {
                            "start_vertex": "h2_MIDAT"
                        },
                        "elec_edge": {
                            "end_vertex": "elec_MIDAT",
                            "existing_capacity": 0,
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.1
                        }
                    }
                },
                {
                    "id": "NE_FuelCell",
                    "transforms": {
                        "efficiency_rate": 0.875111139
                    },
                    "edges": {
                        "h2_edge": {
                            "start_vertex": "h2_NE"
                        },
                        "elec_edge": {
                            "end_vertex": "elec_NE",
                            "existing_capacity": 0,
                            "investment_cost": 41112.53426,
                            "fixed_om_cost": 1052.480877,
                            "variable_om_cost": 0.0,
                            "capacity_size": 1.5752,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.1
                        }
                    }
                }
            ]
        }
    ]
}
```

10.8 Gas Storage

Graph structure

A storage for a gas commodity is represented in MACRO using the following graph structure:

A gas storage asset is made of:

- 1 `Storage` component, representing the gas storage process. The gas type is set using the `commodity` attribute (see table below).
- 1 `Transformation` component, representing the gas compressor.

- 4 Edge components:
  - 1 **incoming** `Electricity` Edge, representing the electricity consumption for powering the compressor.
  - 1 **incoming** `Gas` Edge, representing the gas flow into the storage asset through the compressor.
  - 1 **internal** `Gas` Edge, representing the gas flow between the compressor and the storage. This can be seen as a charge edge for the storage component.
  - 1 **outgoing** `Gas` Edge, representing the discharged edge of the gas storage.

Attributes

The structure of the input file for a gas storage asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "compressor_elec_edge": {
            // ... compressor_elec_edge-specific attributes ...
        },
        "compressor_gas_edge": {
            // ... compressor_gas_edge-specific attributes ...
        },
        "charge_edge": {
            // ... storage_gas_edge-specific attributes ...
        },
        "discharge_edge": {
            // ... discharge_gas_edge-specific attributes ...
        }
    },
    "storage":{
        // ... storage-specific attributes ...
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

GasStorage

Note: c is the type of the commodity being stored. The following equation is related to the compressor.

$$\phi_{elec} = \phi_c \cdot \epsilon_{elec\_consumption}$$

Look also at the "Efficiency" tip below for more information on the efficiency of charging/discharging process.

Edges

All the edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Efficiency

The efficiency of the charging/discharging process can be set in the `charge_edge` and `discharge_edge` parts of the input file. These parameters are used, for example, in the Balance constraint to balance the charge and discharge flows.

Default constraints

The only **default constraint** for the edges of the gas storage asset is the Capacity constraint applied to both the charge and discharge edges.

Storage component

The definition of the `Storage` object can be found here Macro.Storage.

Default constraints

As noted in the above table, the **default constraints** for the storage component of the gas storage are the following:

- Balance constraint
- Storage capacity constraint

Example

The following input file example shows how to create a hydrogen storage asset in each of the three zones SE, MIDAT and NE.

```
{
    "h2stor": [
        {
            "type": "GasStorage",
            "global_data": {
                "nodes": {},
                "transforms": {
                    "timedata": "Hydrogen",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "discharge_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "can_expand": true,
                        "can_retire": false,
                        "has_capacity": true,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true
                        }
                    },
                    "charge_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": true,
                        "can_retire": false,
                        "constraints": {
                            "CapacityConstraint": true
                        }
                    },
                    "compressor_elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "compressor_gas_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": false
                    }
                },
                "storage": {
                    "commodity": "Hydrogen",
                    "can_expand": true,
                    "can_retire": false,
                    "constraints": {
                        "StorageCapacityConstraint": true,
                        "BalanceConstraint": true,
                        "MinStorageLevelConstraint": true
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_Above_ground_storage",
                    "transforms": {
                        "electricity_consumption": 0.018029457
                    },
                    "edges": {
```

```
                "discharge_edge": {
                    "end_vertex": "h2_SE",
                    "existing_capacity": 0,
                    "investment_cost": 0.0,
                    "fixed_om_cost": 0.0,
                    "variable_om_cost": 0.0,
                    "efficiency": 1.0,
                    "ramp_up_fraction": 1,
                    "ramp_down_fraction": 1
                },
                "charge_edge": {
                    "existing_capacity": 0,
                    "investment_cost": 3219.236569,
                    "fixed_om_cost": 0.0,
                    "variable_om_cost": 0.0,
                    "efficiency": 1.0
                },
                "compressor_gas_edge": {
                    "start_vertex": "h2_SE"
                },
                "compressor_elec_edge": {
                    "start_vertex": "elec_SE"
                }
            },
            "storage": {
                "investment_cost_storage": 873.013307,
                "fixed_om_cost_storage": 28.75810056,
                "storage_loss_fraction": 0.0,
                "min_storage_level": 0.3
            }
        },
        {
            "id": "MIDAT_Above_ground_storage",
            "transforms": {
                "electricity_consumption": 0.018029457
            },
            "edges": {
                "discharge_edge": {
                    "end_vertex": "h2_MIDAT",
                    "existing_capacity": 0,
                    "investment_cost": 0.0,
                    "fixed_om_cost": 0.0,
                    "variable_om_cost": 0.0,
                    "efficiency": 1.0,
                    "ramp_up_fraction": 1,
                    "ramp_down_fraction": 1
                },
                "charge_edge": {
                    "existing_capacity": 0,
                    "investment_cost": 3219.236569,
                    "fixed_om_cost": 0.0,
                    "variable_om_cost": 0.0,
                    "efficiency": 1.0
                },
                "compressor_gas_edge": {
```

```
                            "start_vertex": "h2_MIDAT"
                        },
                        "compressor_elec_edge": {
                            "start_vertex": "elec_MIDAT"
                        }
                    },
                    "storage": {
                        "investment_cost_storage": 873.013307,
                        "fixed_om_cost_storage": 28.75810056,
                        "storage_loss_fraction": 0.0,
                        "min_storage_level": 0.3
                    }
                },
                {
                    "id": "NE_Above_ground_storage",
                    "transforms": {
                        "electricity_consumption": 0.018029457
                    },
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "h2_NE",
                            "existing_capacity": 0,
                            "investment_cost": 0.0,
                            "fixed_om_cost": 0.0,
                            "variable_om_cost": 0.0,
                            "efficiency": 1.0,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1
                        },
                        "charge_edge": {
                            "existing_capacity": 0,
                            "investment_cost": 3219.236569,
                            "fixed_om_cost": 0.0,
                            "variable_om_cost": 0.0,
                            "efficiency": 1.0
                        },
                        "compressor_gas_edge": {
                            "start_vertex": "h2_NE"
                        },
                        "compressor_elec_edge": {
                            "start_vertex": "elec_NE"
                        }
                    },
                    "storage": {
                        "investment_cost_storage": 873.013307,
                        "fixed_om_cost_storage": 28.75810056,
                        "storage_loss_fraction": 0.0,
                        "min_storage_level": 0.3
                    }
                }
            ]
        }
    ]
}
```

10.9 Hydrogen Line

Graph structure

A hydrogen line is represented in MACRO using the following graph structure:

A hydrogen line asset is very simple and is made of:

- 1 Edge component:
    - 1 `Hydrogen` Edge, representing the flow of hydrogen between two nodes.

Attributes

The structure of the input file for a hydrogen line asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "edges":{
        "h2_edge": {
            // ... h2_edge-specific attributes ...
        }
    }
}
```

Edge

The definition of the Edge object can be found here Macro.Edge.

Default constraints

The **default constraints** for the hydrogen line asset are the following:

- Capacity constraint

Example

The following is an example of the input file for a hydrogen line asset that creates two hydrogen lines, one connecting the SE and MIDAT regions, and one connecting the NE and SE regions.

```
{
    "h2transport": [
        {
            "type": "HydrogenLine",
            "global_data": {
                "edges": {
                    "h2_edge": {
                        "type": "Hydrogen",
                        "unidirectional": false,
                        "can_expand": true,
                        "can_retire": false,
                        "has_capacity": true,
                        "integer_decisions": false,
                        "constraints": {
                            "CapacityConstraint": true
                        }
                    }
                }
            },
            "instance_data": [
                {
                    "id": "h2_SE_to_MIDAT",
                    "edges": {
                        "h2_edge": {
                            "start_vertex": "h2_SE",
                            "end_vertex": "h2_MIDAT",
                            "loss_fraction": 0.067724471,
                            "distance": 491.4512001,
                            "capacity_size": 787.6,
                            "investment_cost": 82682.23402
                        }
                    }
                },
                {
                    "id": "h2_NE_to_SE",
                    "edges": {
                        "h2_edge": {
                            "start_vertex": "h2_NE",
                            "end_vertex": "h2_SE",
                            "loss_fraction": 0.06553874,
                            "distance": 473.6625536,
                            "capacity_size": 787.6,
                            "investment_cost": 79896.9841
                        }
                    }
                }
            ]
        }
    ]
}
```

10.10 Hydro Reservoir

Graph structure

A hydroelectric reservoir is represented in MACRO using the following graph structure:

A hydroelectric reservoir asset is made of:

- 1 `Storage` component, representing the hydroelectric reservoir.
- 3 Edge components:
  - 1 **incoming** `Electricity` Edge, representing the electricity supply.
  - 1 **outgoing** `Electricity` Edge, representing the electricity production.
  - 1 **outgoing** `Electricity` Edge, representing the spillage.

Flow equation

In the following equation, ϕ is the flow of the commodity.

HydroRes

$$\phi_{in} = \phi_{out} + \phi_{spill}$$

Attributes

The structure of the input file for a hydroelectric reservoir asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "inflow_edge": {
            // ... inflow_edge-specific attributes ...
        },
        "discharge_edge": {
            // ... discharge_edge-specific attributes ...
        },
        "spillage_edge": {
            // ... spillage_edge-specific attributes ...
        }
    }
}
```

Storage component

The definition of the `Storage` object can be found here Macro.Storage.

Default constraints

The **default constraints** for the storage component of the hydroelectric reservoir are the following:

- Balance constraint

Edges (discharge_edge, inflow_edge, spillage_edge)

Asset expansion

As a modeling decision, only charge and discharge edges are allowed to expand. Therefore, the `has_capacity` attribute can only be set for the `discharge_edge` and `inflow_edge`. For the spillage edge, this attribute is pre-set to `false` to ensure the correct modeling of the asset.

Directionality

All the three edges are unidirectional by construction.

All the edges have the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Default constraints

Default constraints for the edges of the hydroelectric reservoir are only applied to the inflow edge. These constraints are:

- Must run constraint
- Storage charge discharge ratio constraint

Example

The following input file example shows how to create a hydroelectric reservoir asset in each of the three zones SE, MIDAT and NE.

```
{
    "hydrores": [
        {
            "type": "HydroRes",
            "global_data": {
                "edges": {
                    "discharge_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": false,
                        "can_retire": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true
                        }
                    },
                    "inflow_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "start_vertex": "hydro_source",
                        "has_capacity": true,
                        "can_expand": false,
                        "can_retire": false,
                        "constraints": {
                            "MustRunConstraint": true
                        }
                    },
                    "spill_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "end_vertex": "hydro_source",
                        "can_expand": false,
                        "can_retire": false,
                        "has_capacity": false
                    }
                },
                "storage": {
                    "commodity": "Electricity",
                    "can_expand": false,
                    "can_retire": false,
                    "constraints": {
                        "MinStorageOutflowConstraint": true,
                        "StorageChargeDischargeRatioConstraint": true,
                        "BalanceConstraint": true
                    }
                }
            },
            "instance_data": [
                {
                    "id": "MIDAT_conventional_hydroelectric_1",
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "elec_MIDAT",
                            "capacity_size": 29.853,
                            "existing_capacity": 2806.182,
                            "fixed_om_cost": 45648,
                            "ramp_down_fraction": 0.83,
                            "ramp_up_fraction": 0.83,
                            "efficiency": 1.0
                        },
                        "inflow_edge": {
                            "efficiency": 1.0,
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_conventional_hydroelectric_1"
                                }
                            }
                        }
                    },
                    "storage": {
                        "min_outflow_fraction": 0.109311313,
                        "charge_discharge_ratio": 1.0
                    }
                },
                {
                    "id": "NE_conventional_hydroelectric_1",
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "elec_NE",
                            "capacity_size": 24.13,
                            "existing_capacity": 4729.48,
                            "fixed_om_cost": 45648,
                            "ramp_down_fraction": 0.083,
                            "ramp_up_fraction": 0.083,
                            "efficiency": 1.0
                        },
                        "inflow_edge": {
                            "efficiency": 1.0,
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_conventional_hydroelectric_1"
                                }
                            }
                        }
                    },
                    "storage": {
                        "min_outflow_fraction": 0.095,
                        "charge_discharge_ratio": 1.0
                    }
                },
                {
                    "id": "SE_conventional_hydroelectric_1",
                    "edges": {
                        "discharge_edge": {
                            "end_vertex": "elec_SE",
                            "capacity_size": 31.333,
                            "existing_capacity": 11123.215,
                            "fixed_om_cost": 45648,
                            "ramp_down_fraction": 0.083,
                            "ramp_up_fraction": 0.083,
                            "efficiency": 1.0
                        },
                        "inflow_edge": {
                            "efficiency": 1.0,
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_conventional_hydroelectric_1"
                                }
                            }
                        }
                    },
                    "storage": {
                        "min_outflow_fraction": 0.135129141,
                        "charge_discharge_ratio": 1.0
                    }
                }
            ]
        }
    ]
}
```

10.11 Must Run

Graph structure

A MustRun asset is represented in MACRO using the following graph structure:

A MustRun asset is very similar to a VRE asset, and is made of:

- 1 `Transformation` component, representing the MustRun transformation.

- 1 Edge component:
  - 1 **outgoing** `Electricity` Edge, representing the electricity production.

Attributes

The structure of the input file for a hydroelectric reservoir asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Edges

The definition of the Edge object can be found here Macro.Edge.

Default constraints

Default constraints for the edges of the MustRun asset are only applied to the inflow edge. These constraints are:

- Must run constraint

Example

The following input file example shows how to create a MustRun asset in each of the three zones SE, MIDAT and NE.

```
{
    "mustrun": [
        {
            "type": "MustRun",
            "global_data": {
                "nodes": {},
                "transforms": {
                    "timedata": "Electricity"
                },
                "edges": {
                    "elec_edge": {
                        "unidirectional": true,
                        "can_expand": false,
                        "can_retire": false,
                        "has_capacity": true,
                        "constraints": {
                            "MustRunConstraint": true
                        }
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_small_hydroelectric_1",
                    "edges": {
                        "elec_edge": {
                            "end_vertex": "elec_SE",
                            "existing_capacity": 249.895,
                            "capacity_size": 1.219,
                            "fixed_om_cost": 45648,
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_small_hydroelectric_1"
                                }
                            }
                        }
                    }
                },
                {
                    "id": "MIDAT_small_hydroelectric_1",
                    "edges": {
                        "elec_edge": {
                            "end_vertex": "elec_MIDAT",
                            "existing_capacity": 263.268,
                            "capacity_size": 1.236,
                            "fixed_om_cost": 45648,
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_small_hydroelectric_1"
                                }
                            }
                        }
                    }
                },
                {
                    "id": "NE_small_hydroelectric_1",
                    "edges": {
                        "elec_edge": {
                            "end_vertex": "elec_NE",
                            "existing_capacity": 834.494,
                            "capacity_size": 1.051,
                            "fixed_om_cost": 45648,
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_small_hydroelectric_1"
                                }
                            }
                        }
                    }
                }
            ]
        }
    ]
}
```

10.12 Natural Gas DAC

Graph structure

A natural gas DAC is represented in MACRO using the following graph structure:

A natural gas DAC asset is made of:

- 1 `Transformation` component, representing the natural gas DAC process.
- 5 Edge components:
  - 1 **incoming** `NaturalGas` Edge, representing the natural gas supply.
  - 1 **incoming** `CO2` Edge, representing the CO2 that is absorbed by the natural gas DAC process.
  - 1 **outgoing** `Electricity` Edge, representing the electricity production.
  - 1 **outgoing** `CO2Captured` Edge, representing the CO2 that is captured.
  - 1 **outgoing** `CO2` Edge, representing the CO2 that is emitted.

Attributes

The structure of the input file for a natural gas DAC asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "ng_edge": {
            // ... ng_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        },
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "co2_emission_edge": {
            // ... co2_emission_edge-specific attributes ...
        },
        "co2_captured_edge": {
            // ... co2_captured_edge-specific attributes ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the natural gas DAC asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

NaturalGasDAC

$$\begin{aligned}
\phi_{elec} &= \phi_{co2} \cdot \epsilon_{elec\_prod} \\
\phi_{ng} &= -\phi_{co2} \cdot \epsilon_{fuel\_consumption} \\
\phi_{co2} &= \phi_{ng} \cdot \epsilon_{emission\_rate} \\
\phi_{co2\_captured} + \phi_{co2} &= \phi_{ng} \cdot \epsilon_{co2\_capture\_rate}
\end{aligned}$$

Edges

Asset expansion

As a modeling decision, only the incoming `CO2` edge is allowed to expand. Therefore, the `has_capacity` attribute can only be set for this edge. For all the other edges, this attribute is pre-set to `false` to ensure the correct modeling of the asset.

Directionality

The `unidirectional` attribute is only available for the incoming `CO2` edge. For the other edges, this attribute is pre-set to `true` to ensure the correct modeling of the asset.

The definition of the Edge object can be found here Macro.Edge.

Default constraints

The only **default constraint** for the edges of the natural gas DAC asset is the Capacity constraint applied to the incoming `CO2` edge.

Example

The following input file example shows how to create a natural gas DAC asset in each of the three zones NE, SE and MIDAT.

```
{
    "NaturalGasDAC": [
        {
            "type": "NaturalGasDAC",
            "global_data": {
                "transforms": {
                    "timedata": "NaturalGas",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": true,
                        "start_vertex": "co2_sink",
                        "can_retire": true,
                        "can_expand": true,
                        "integer_decisions": false,
                        "uc": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true
                        }
                    },
                    "co2_emission_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_sink"
                    },
                    "ng_edge": {
                        "type": "NaturalGas",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "co2_captured_edge": {
                        "type": "CO2Captured",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_captured_sink"
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_Sorbent_DAC",
                    "transforms": {
                        "emission_rate": 0.005516648,
                        "capture_rate": 0.546148172,
                        "electricity_production": 0.125,
                        "fuel_consumption": 3.047059915
                    },
                    "edges": {
                        "co2_edge": {
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_Sorbent_DAC"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 869000.00,
                            "fixed_om_cost": 384000.00,
                            "variable_om_cost": 58.41,
                            "ramp_up_fraction": 1.0,
                            "ramp_down_fraction": 1.0
                        },
                        "ng_edge": {
                            "start_vertex": "natgas_SE"
                        },
                        "elec_edge": {
                            "end_vertex": "elec_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_Sorbent_DAC",
                    "transforms": {
                        "emission_rate": 0.005516648,
                        "capture_rate": 0.546148172,
                        "electricity_production": 0.125,
                        "fuel_consumption": 3.047059915
                    },
                    "edges": {
                        "co2_edge": {
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_Sorbent_DAC"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 869000.00,
                            "fixed_om_cost": 384000.00,
                            "variable_om_cost": 58.41,
                            "ramp_up_fraction": 1.0,
                            "ramp_down_fraction": 1.0
                        },
                        "ng_edge": {
                            "start_vertex": "natgas_MIDAT"
                        },
                        "elec_edge": {
                            "end_vertex": "elec_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_Sorbent_DAC",
                    "transforms": {
                        "emission_rate": 0.005516648,
                        "capture_rate": 0.546148172,
                        "electricity_production": 0.125,
                        "fuel_consumption": 3.047059915
                    },
                    "edges": {
                        "co2_edge": {
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_Sorbent_DAC"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 869000.00,
                            "fixed_om_cost": 384000.00,
                            "variable_om_cost": 58.41,
                            "ramp_up_fraction": 1.0,
                            "ramp_down_fraction": 1.0
                        },
                        "ng_edge": {
                            "start_vertex": "natgas_NE"
                        },
                        "elec_edge": {
                            "end_vertex": "elec_NE"
                        }
                    }
                }
            ]
        }
    ]
}
```

10.13 Power Line

Graph structure

A power line is represented in MACRO using the following graph structure:

A power line asset is very simple and is made of:

- 1 Edge component:
    - 1 `Electricity` Edge, representing the flow of electricity between two nodes.

Attributes

The structure of the input file for a power line asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "edges":{
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        }
    }
}
```

Edges

The definition of the Edge object can be found here Macro.Edge.

Default constraints

The **default constraint** for power lines is the Capacity constraint.

Example

The following is an example of the input file for a power line asset that creates two power lines, one connecting the SE and MIDAT regions, and one connecting the MIDAT and NE regions.

```
{
    "line": [
        {
            "type": "PowerLine",
            "global_data": {
                "edges": {
                    "elec_edge": {
                        "type": "Electricity",
                        "has_capacity": true,
                        "unidirectional": false,
                        "can_expand": true,
                        "can_retire": false,
                        "integer_decisions": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "MaxCapacityConstraint": true
                        }
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_to_MIDAT",
                    "edges": {
                        "elec_edge": {
                            "start_vertex": "elec_SE",
                            "end_vertex": "elec_MIDAT",
                            "distance": 491.4512001,
                            "existing_capacity": 5552,
                            "max_capacity": 33312,
                            "investment_cost": 35910,
                            "loss_fraction": 0.04914512
                        }
                    }
                },
                {
                    "id": "MIDAT_to_NE",
                    "edges": {
                        "elec_edge": {
                            "start_vertex": "elec_MIDAT",
                            "end_vertex": "elec_NE",
                            "distance": 473.6625536,
                            "existing_capacity": 1915,
                            "max_capacity": 11490,
                            "investment_cost": 55639,
                            "loss_fraction": 0.047366255
                        }
                    }
                }
            ]
        }
    ]
}
```

10.14 Thermal Hydrogen Plant (with and without CCS)

Graph structure

A thermal hydrogen plant (with and without CCS) is represented in MACRO using the following graph structure:

A thermal hydrogen plant (with and without CCS) is made of:

- 1 `Transformation` component, representing the thermal hydrogen plant (with and without CCS).
- 5 Edge components:
  - 1 **incoming** `Fuel` Edge, representing the fuel supply.
  - 1 **incoming** `Electricity` Edge, representing the electricity consumption.
  - 1 **outgoing** `Hydrogen` Edge, representing the hydrogen production. **This edge can have unit commitment operations**.
  - 1 **outgoing** `CO2` Edge, representing the CO2 that is emitted.
  - 1 **outgoing** `CO2Captured` Edge, representing the CO2 that is captured **(only if CCS is present)**.

Attributes

The structure of the input file for a ThermalHydrogen asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "fuel_edge": {
            // ... fuel_edge-specific attributes ...
        },
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "h2_edge": {
            // ... h2_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        },
        "co2_captured_edge": {
            // ... co2_captured_edge-specific attributes, only if CCS is present ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the ThermalHydrogen asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

ThermalHydrogen

Note: Fuel is the type of the fuel being converted.

$$\begin{aligned}
\phi_{h2} &= \phi_{fuel} \cdot \epsilon_{efficiency} \\
\phi_{elec} &= \phi_{h2} \cdot \epsilon_{elec\_consumption} \\
\phi_{co2} &= \phi_{fuel} \cdot \epsilon_{emission\_rate} \\
\phi_{co2\_captured} &= \phi_{fuel} \cdot \epsilon_{co2\_capture\_rate} \quad \text{(if CCS)}
\end{aligned}$$

Edges

Asset expansion

As a modeling decision, only the Hydrogen and Fuel edges are allowed to expand. Therefore, both the has_capacity and constraints attributes can only be set for those edges. For all the other edges, these attributes are pre-set to false and to an empty list respectively to ensure the correct modeling of the asset.

Directionality

The unidirectional attribute is set to true for all the edges.

Unit commitment and default constraints

The Hydrogen edge **can have unit commitment operations**. To enable it, the user needs to set the uc attribute to true. The default constraints for unit commitment case are the following:

- Capacity constraint
- Ramping limits constraint
- Minimum up and down time constraint

In case of no unit commitment, the uc attribute is set to false and the default constraints are the following:

- Capacity constraint

All the edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Example

The following is an example of the input file for a ThermalHydrogenCCS asset that creates three ThermalHydrogenCCS assets, one in each of the SE, MIDAT and NE regions.

```
{
    "NaturalGasH2ATRCCS": [
        {
            "type": "ThermalHydrogenCCS",
            "global_data": {
                "transforms": {
                    "timedata": "NaturalGas",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "h2_edge": {
                        "type": "Hydrogen",
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_retire": true,
                        "can_expand": true,
                        "integer_decisions": false,
                        "uc": true,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true,
                            "MinFlowConstraint": true,
                            "MinUpTimeConstraint": true,
                            "MinDownTimeConstraint": true
                        }
                    },
                    "fuel_edge": {
                        "type": "NaturalGas",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_sink"
                    },
                    "co2_captured_edge": {
                        "type": "CO2Captured",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_captured_sink"
                    },
                    "elec_edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "has_capacity": false
                    }
                }
            },
            "instance_data": [
                {
                    "id": "SE_ATR_wCCS_94pct",
                    "transforms": {
                        "emission_rate": 0.003794329,
                        "efficiency_rate": 0.769121482,
                        "electricity_consumption": 0.101574,
                        "capture_rate": 0.065193472
                    },
                    "edges": {
                        "h2_edge": {
                            "end_vertex": "h2_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_ATR_wCCS_94pct"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 57497.91679,
                            "fixed_om_cost": 23292.27286,
                            "variable_om_cost": 9.262366684,
                            "capacity_size": 1082.95,
                            "startup_cost": 0.253936008,
                            "min_up_time": 22,
                            "min_down_time": 12,
                            "ramp_up_fraction": 0.5,
                            "ramp_down_fraction": 0.5,
                            "min_flow_fraction": 0.85
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_SE"
                        },
                        "elec_edge": {
                            "start_vertex": "elec_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_ATR_wCCS_94pct",
                    "transforms": {
                        "emission_rate": 0.003794329,
                        "efficiency_rate": 0.769121482,
                        "electricity_consumption": 0.101574,
                        "capture_rate": 0.065193472
                    },
                    "edges": {
                        "h2_edge": {
                            "end_vertex": "h2_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_ATR_wCCS_94pct"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 57497.91679,
                            "fixed_om_cost": 23292.27286,
                            "variable_om_cost": 9.262366684,
                            "capacity_size": 1082.95,
                            "startup_cost": 0.253936008,
                            "min_up_time": 22,
                            "min_down_time": 12,
                            "ramp_up_fraction": 0.5,
                            "ramp_down_fraction": 0.5,
                            "min_flow_fraction": 0.85
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_MIDAT"
                        },
                        "elec_edge": {
                            "start_vertex": "elec_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_ATR_wCCS_94pct",
                    "transforms": {
                        "emission_rate": 0.003794329,
                        "efficiency_rate": 0.769121482,
                        "electricity_consumption": 0.101574,
                        "capture_rate": 0.065193472
                    },
                    "edges": {
                        "h2_edge": {
                            "end_vertex": "h2_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_ATR_wCCS_94pct"
                                }
                            },
                            "existing_capacity": 0.0,
                            "investment_cost": 57497.91679,
                            "fixed_om_cost": 23292.27286,
                            "variable_om_cost": 9.262366684,
                            "capacity_size": 1082.95,
                            "startup_cost": 0.253936008,
                            "min_up_time": 22,
                            "min_down_time": 12,
                            "ramp_up_fraction": 0.5,
                            "ramp_down_fraction": 0.5,
                            "min_flow_fraction": 0.85
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_NE"
                        },
                        "elec_edge": {
                            "start_vertex": "elec_NE"
                        }
                    }
                }
            ]
        }
    ]
}
```

10.15 Thermal Power Plant (with and without CCS)

Graph structure

A thermal power plant (with and without CCS) is represented in MACRO using the following graph structure:

A thermal power plant (with and without CCS) is made of:

- 1 `Transformation` component, representing the thermal power plant (with and without CCS).
- 4 Edge components:
  - 1 **incoming** `Fuel` Edge, representing the fuel supply.
  - 1 **outgoing** `Electricity` Edge, representing the electricity production. **This edge can have unit commitment operations**.
  - 1 **outgoing** `CO2` Edge, representing the CO2 that is emitted.
  - 1 **outgoing** `CO2Captured` Edge, representing the CO2 that is captured **(only if CCS is present)**.

Attributes

The structure of the input file for a ThermalPower asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "fuel_edge": {
            // ... fuel_edge-specific attributes ...
        },
        "elec_edge": {
            // ... elec_edge-specific attributes ...
        },
        "co2_edge": {
            // ... co2_edge-specific attributes ...
        },
        "co2_captured_edge": {
            // ... co2_captured_edge-specific attributes, only if CCS is present ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Default constraints

The **default constraint** for the transformation part of the thermal power asset is the following:

- Balance constraint

Flow equations

In the following equations, ϕ is the flow of the commodity and ϵ is the stoichiometric coefficient defined in the transformation table below.

ThermalPower

Note: Fuel is the type of the fuel being converted.

$$\phi_{elec} = \phi_{fuel} \cdot \epsilon_{efficiency}$$
$$\phi_{co2} = \phi_{fuel} \cdot \epsilon_{emission\_rate}$$
$$\phi_{co2\_captured} = \phi_{fuel} \cdot \epsilon_{co2\_capture\_rate} \quad \text{(if CCS)}$$

Edges

Asset expansion

As a modeling decision, only the `Electricity` and `Fuel` edges are allowed to expand. Therefore, both the `has_capacity` and `constraints` attributes can only be set for those edges. For all the other edges, these attributes are pre-set to `false` and to an empty list respectively to ensure the correct modeling of the asset.

Directionality

The `unidirectional` attribute is set to `true` for all the edges.

Unit commitment and default constraints

The `Electricity` edge **can have unit commitment operations**. To enable it, the user needs to set the `uc` attribute to `true`. The default constraints for unit commitment case are the following:

- Capacity constraint
- Ramping limits constraint
- Minimum up and down time constraint

In case of no unit commitment, the `uc` attribute is set to `false` and the default constraints are the following:

- Capacity constraint

All the edges are represented by the same set of attributes. The definition of the Edge object can be found here Macro.Edge.

Example

The following is an example of the input file for a ThermalPowerCCS asset that creates three ThermalPowerCCS assets, one in each of the SE, MIDAT and NE regions.

```
{
    "NaturalGasPowerCCS": [
        {
            "type": "ThermalPowerCCS",
            "global_data": {
                "transforms": {
                    "timedata": "NaturalGas",
                    "constraints": {
                        "BalanceConstraint": true
                    }
                },
                "edges": {
                    "elec_edge": {
                        "type": "Electricity",
                        "uc": true,
                        "unidirectional": true,
                        "has_capacity": true,
                        "can_expand": true,
                        "can_retire": true,
                        "integer_decisions": false,
                        "constraints": {
                            "CapacityConstraint": true,
                            "RampingLimitConstraint": true,
                            "MinFlowConstraint": true,
                            "MinUpTimeConstraint": true,
                            "MinDownTimeConstraint": true
                        }
                    },
                    "fuel_edge": {
                        "type": "NaturalGas",
                        "unidirectional": true,
                        "has_capacity": false
                    },
                    "co2_edge": {
                        "type": "CO2",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_sink"
                    },
                    "co2_captured_edge": {
                        "type": "CO2Captured",
                        "unidirectional": true,
                        "has_capacity": false,
                        "end_vertex": "co2_captured_sink"
                    }
                },
            },
            "instance_data": [
                {
                    "id": "SE_naturalgas_ccccsavgcf_conservative_0",
                    "transforms": {
                        "efficiency_rate": 0.476622662,
                        "emission_rate": 0.018104824,
                        "capture_rate": 0.162943412
                    },
                    "edges": {
                        "elec_edge": {
                            "end_vertex": "elec_SE",
                            "investment_cost": 150408.6558,
                            "existing_capacity": 0.0,
                            "fixed_om_cost": 65100,
                            "variable_om_cost": 5.73,
                            "capacity_size": 377,
                            "startup_cost": 97,
                            "startup_fuel": 0.058614214,
                            "min_up_time": 4,
                            "min_down_time": 4,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.5
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_SE"
                        }
                    }
                },
                {
                    "id": "MIDAT_naturalgas_ccccsavgcf_conservative_0",
                    "transforms": {
                        "efficiency_rate": 0.476622662,
                        "emission_rate": 0.018104824,
                        "capture_rate": 0.162943412
                    },
                    "edges": {
                        "elec_edge": {
                            "end_vertex": "elec_MIDAT",
                            "investment_cost": 158946.1077,
                            "existing_capacity": 0.0,
                            "fixed_om_cost": 65100,
                            "variable_om_cost": 5.73,
                            "capacity_size": 377,
                            "startup_cost": 97,
                            "startup_fuel": 0.058614214,
                            "min_up_time": 4,
                            "min_down_time": 4,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.5
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_MIDAT"
                        }
                    }
                },
                {
                    "id": "NE_naturalgas_ccccsavgcf_conservative_0",
                    "transforms": {
                        "efficiency_rate": 0.476622662,
                        "emission_rate": 0.018104824,
                        "capture_rate": 0.162943412
                    },
                    "edges": {
                        "elec_edge": {
                            "end_vertex": "elec_NE",
                            "investment_cost": 173266.9946,
                            "existing_capacity": 0.0,
                            "fixed_om_cost": 65100,
                            "variable_om_cost": 5.73,
                            "capacity_size": 377,
                            "startup_cost": 97,
                            "startup_fuel": 0.058614214,
                            "min_up_time": 4,
                            "min_down_time": 4,
                            "ramp_up_fraction": 1,
                            "ramp_down_fraction": 1,
                            "min_flow_fraction": 0.5
                        },
                        "fuel_edge": {
                            "start_vertex": "natgas_NE"
                        }
                    }
                }
            ]
        }
    ]
}
```

10.16 Variable Renewable Energy resources (VRE)

Graph structure

A Variable Renewable Energy asset is represented in MACRO using the following graph structure:

A Variable Renewable Energy asset is made of:

- 1 `Transformation` component, representing the VRE transformation.
- 1 Edge component:
    - 1 **outgoing** `Electricity` Edge, representing the electricity production.

Attributes

The structure of the input file for a VRE asset follows the graph representation. Each `global_data` and `instance_data` will look like this:

```
{
    "transforms":{
        // ... transformation-specific attributes ...
    },
    "edges":{
        "edge": {
            // ... electricity edge-specific attributes ...
        }
    }
}
```

Transformation

The definition of the transformation object can be found here Macro.Transformation.

Edges

The definition of the Edge object can be found here Macro.Edge.

Default constraint

Default constraint for the electricity edge of the VRE is the Capacity constraint.

Example

The following input file example shows how to create four existing VRE assets (two utility-scale solar and two onshore wind facilities) and four new VRE assets (one offshore wind, one onshore wind, and two utility-scale solar facilities).

```
{
    "existing_vre": [
        {
            "type": "VRE",
            "global_data": {
                "transforms": {
                    "timedata": "Electricity"
                },
                "edges": {
                    "edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "can_expand": false,
                        "can_retire": true,
                        "has_capacity": true,
                        "constraints": {
                            "CapacityConstraint": true
                        }
                    }
                },
                "storage": {}
            },
            "instance_data": [
                {
                    "id": "existing_solar_SE",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 22887,
                            "capacity_size": 17.142,
                            "existing_capacity": 8502.2,
                            "end_vertex": "elec_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_solar_photovoltaic_1"
                                }
                            }
                        }
                    },
                },
                {
                    "id": "existing_solar_NE",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 22887,
                            "capacity_size": 3.63,
                            "existing_capacity": 1629.6,
                            "end_vertex": "elec_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_solar_photovoltaic_1"
                                }
                            }
                        }
                    },
                },
                {

                    "id": "existing_wind_NE",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 43000,
                            "capacity_size": 86.17,
                            "existing_capacity": 3654.5,
                            "end_vertex": "elec_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_onshore_wind_turbine_1"
                                }
                            }
                        }
                    },
                },
                {
                    "id": "existing_wind_MIDAT",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 43000,
                            "capacity_size": 161.2,
                            "existing_capacity": 3231.6,
                            "end_vertex": "elec_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_onshore_wind_turbine_1"
                                }
                            }
                        }
                    },
                }
            ]
```

```
        }
    ],
    "new_vre": [
        {
            "type": "VRE",
            "global_data": {
                "transforms": {
                    "timedata": "Electricity"
                },
                "edges": {
                    "edge": {
                        "type": "Electricity",
                        "unidirectional": true,
                        "can_expand": true,
                        "can_retire": false,
                        "has_capacity": true,
                        "constraints": {
                            "CapacityConstraint": true,
                            "MaxCapacityConstraint": true
                        }
                    }
                },
            },
            "instance_data": [
                {
                    "id": "NE_offshorewind_class10_moderate_floating_1_1",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 56095.98976,
                            "investment_cost": 225783.4407,
                            "max_capacity": 32928.493,
                            "end_vertex": "elec_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_offshorewind_class10_moderate_floating_1_1"
                                }
                            }
                        }
                    },
                },
                {
                    "id": "SE_utilitypv_class1_moderate_70_0_2_1",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 15390.48615,
                            "investment_cost": 49950.17548,
                            "max_capacity": 1041244,
                            "end_vertex": "elec_SE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "SE_utilitypv_class1_moderate_70_0_2_1"
                                }
                            }
                        }
                    },
                },
                {
                    "id": "MIDAT_utilitypv_class1_moderate_70_0_2_1",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 15390.48615,
                            "investment_cost": 51590.03227,
                            "max_capacity": 26783,
                            "end_vertex": "elec_MIDAT",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "MIDAT_utilitypv_class1_moderate_70_0_2_1"
                                }
                            }
                        }
                    },
                },
                {
                    "id": "NE_landbasedwind_class4_moderate_70_3",
                    "edges": {
                        "edge": {
                            "fixed_om_cost": 34568.125,
                            "investment_cost": 86536.01624,
                            "max_capacity": 65324,
                            "end_vertex": "elec_NE",
                            "availability": {
                                "timeseries": {
                                    "path": "assets/availability.csv",
                                    "header": "NE_landbasedwind_class4_moderate_70_3"
                                }
                            }
                        }
                    }
                }
            ]
        }
    ]
}
```

Attribute	Type	Values	Default	Description
commodity	`String`	`Electricity`	Required	Commodity being stored.
constraints	`Dict{String,Bool}`	Any MACRO constraint type for storage	`BalanceConstraint`, `StorageCapacityConstraint`, `StorageMaxDurationConstraint`, `StorageMinDurationConstraint`, `StorageSymmetricCapacityConstraint`	List of constraints applied to the storage. E.g. `{"BalanceConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the storage is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the storage is eligible for capacity retirement.
charge_discharge_ratio	`Float64`	`Float64`	`1.0`	Ratio between charging and discharging rates.
existing_capacity_storage	`Float64`	`Float64`	`0.0`	Initial installed storage capacity (MWh).
fixed_om_cost_storage	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MWh-year).
investment_cost_storage	`Float64`	`Float64`	`0.0`	Annualized investment cost of the energy capacity for a storage technology (USD/MWh-year).
max_capacity_storage	`Float64`	`Float64`	`Inf`	Maximum allowed storage capacity (MWh).
max_duration	`Float64`	`Float64`	`0.0`	Maximum ratio of installed energy to discharged capacity that can be installed (hours).
min_capacity_storage	`Float64`	`Float64`	`0.0`	Minimum allowed storage capacity (MWh).
min_duration	`Float64`	`Float64`	`0.0`	Minimum ratio of installed energy to discharged capacity that can be installed (hours).
min_outflow_fraction	`Float64`	`Float64`	`0.0`	Minimum outflow as a fraction of capacity.
min_storage_level	`Float64`	`Float64`	`0.0`	Minimum storage level as a fraction of capacity.
max_storage_level	`Float64`	`Float64`	`1.0`	Maximum storage level as a fraction of capacity.
storage_loss_fraction	`Float64`	Number $\in$ [0,1]	`0.0`	Fraction of stored commodity lost per timestep

Attribute	Type	Values	Default	Description
type	`String`	`Electricity`	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any electricity node id present in the system	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any electricity node id present in the system	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
efficiency	`Float64`	Number ∈ [0,1]	`1.0`	Efficiency of the charging/discharging process.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
ramp_up_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum increase in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
unidirectional	`Bool`	`Bool`	`false`	Whether the edge is unidirectional.
variable_om_cost	`Float64`	`Float64`	`0.0`	Variable operation and maintenance cost (USD/MWh).

Attribute	**Type**	**Values**	**Default**	**Description/Units**
timedata	`String`	Any MACRO commodity type	Required	Time resolution for the time series data linked to the transformation. E.g. "Biomass".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
capture_rate $\epsilon_{co2\_capture\_rate}$	`Float64`	`Float64`	1.0	$t_{CO2}/t_{Biomass}$
co2_content ϵ_{co2}	`Float64`	`Float64`	0.0	$t_{CO2}/t_{Biomass}$
electricity_production ϵ_{elec_prod}	`Float64`	`Float64`	1.0	$MWh_{elec}/t_{Biomass}$
emission_rate $\epsilon_{emission\_rate}$	`Float64`	`Float64`	1.0	$t_{CO2}/t_{Biomass}$

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge **(only available for the `Biomass` edge)**.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_frac-	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps

Attribute	Type	Values	Default	Description/Units
timedata	`String`	Any MACRO commodity type	Required	Time resolution for the time series data linked to the transformation. E.g. "Biomass".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
capture_rate $\epsilon_{co2\_capture\_rate}$	`Float64`	`Float64`	1.0	$t_{CO2}/t_{Biomass}$
co2_content ϵ_{co2}	`Float64`	`Float64`	0.0	$t_{CO2}/t_{Biomass}$
electricity_consumption $\epsilon_{elec\_consumption}$	`Float64`	`Float64`	0.0	$MWh_{elec}/t_{Biomass}$
emission_rate $\epsilon_{emission\_rate}$	`Float64`	`Float64`	1.0	$t_{CO2}/t_{Biomass}$
hydrogen_production ϵ_{h2_prod}	`Float64`	`Float64`	0.0	$t_{H2}/t_{Biomass}$

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge. **(only available for the `Biomass` edge)**
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_frac-	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps

Attribute	Type	Values	Default	Description/Units
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "Electricity".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
electricity_consumption $\epsilon_{elec\_consumption}$	`Float64`	`Float64`	0.0	MWh_{elec}/t_{CO2}

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_frac	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps

Attribute	Type	Values	Default	Description/Units
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "Hydrogen".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Required	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
efficiency_rate $\epsilon_{efficiency}$	`Float64`	`Float64`	1.0	MWh_{h2}/MWh_{elec}

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_frac-	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps

Attribute	**Type**	**Values**	**Default**	**Description/Units**
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "Hydrogen".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
efficiency_rate $\epsilon_{efficiency}$	`Float64`	`Float64`	`1.0`	MWh_{elec}/MWh_{h2}

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_frac-	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps

Attribute	Type	Values	Default	Description
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "Hydrogen".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
electricity_consumption $\epsilon_{elec\_consumption}$	`Float64`	`Float64`	`0.0`	MWh_{elec}/MWh_{gas}

Attribute	**Type**	**Values**	**Default**	**Description**
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
efficiency	`Float64`	Number ∈ [0,1]	`1.0`	Efficiency of the charging/discharging process.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this**

Attribute	Type	Values	Default	Description
commodity	`String`	Any MACRO commodity type	Required	Commodity being stored. E.g. "Hydrogen".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for storage	`BalanceConstraint`, `StorageCapacityConstraint`	List of constraints applied to the storage. E.g. `{"BalanceConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the storage is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the storage is eligible for capacity retirement.
charge_discharge_ratio	`Float64`	`Float64`	`1.0`	Ratio between charging and discharging rates.
existing_capacity_storage	`Float64`	`Float64`	`0.0`	Initial installed storage capacity (MWh).
fixed_om_cost_storage	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MWh-year).
investment_cost_storage	`Float64`	`Float64`	`0.0`	Annualized investment cost of the energy capacity for a storage technology (USD/MWh-year).
max_capacity_storage	`Float64`	`Float64`	`Inf`	Maximum allowed storage capacity (MWh).
max_duration	`Float64`	`Float64`	`0.0`	Maximum ratio of installed energy to discharged capacity that can be installed (hours).
min_capacity_storage	`Float64`	`Float64`	`0.0`	Minimum allowed storage capacity (MWh).
min_duration	`Float64`	`Float64`	`0.0`	Minimum ratio of installed energy to discharged capacity that can be installed (hours).
min_outflow_fraction	`Float64`	`Float64`	`0.0`	Minimum outflow as a fraction of capacity.
min_storage_level	`Float64`	`Float64`	`0.0`	Minimum storage level as a fraction of capacity.
max_storage_level	`Float64`	`Float64`	`1.0`	Maximum storage level as a fraction of capacity.
storage_loss_fraction	`Float64`	Number $\in$ [0,1]	`0.0`	Fraction of stored commodity lost per timestep.

Attribute	Type	Values	Default	Description
type	`String`	`Hydrogen`	Required	Commodity flowing through the edge.
start_vertex	`String`	Any hydrogen node id present in the system	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "h2_node_1".
end_vertex	`String`	Any hydrogen node id present in the system	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "h2_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
distance	`Float64`	`Float64`	`0.0`	Distance between the start and end vertex of the edge.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
ramp_up_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum increase in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
variable_om_cost	`Float64`	`Float64`	`0.0`	Variable operation and maintenance cost (USD/MWh).

Attribute	Type	Values	Default	Description
commodity	`String`	`Electricity`	Required	Commodity being stored.
constraints	`Dict{String,Bool}`	Any MACRO constraint type for storage	`BalanceConstraint`, `StorageCapacityConstraint`	List of constraints applied to the storage. E.g. `{"BalanceConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the storage is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the storage is eligible for capacity retirement.
charge_discharge_ratio	`Float64`	`Float64`	`1.0`	Ratio between charging and discharging rates.
existing_capacity_storage	`Float64`	`Float64`	`0.0`	Initial installed storage capacity (MWh).
fixed_om_cost_storage	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MWh-year).
investment_cost_storage	`Float64`	`Float64`	`0.0`	Annualized investment cost of the energy capacity for a storage technology (USD/MWh-year).
max_capacity_storage	`Float64`	`Float64`	`Inf`	Maximum allowed storage capacity (MWh).
max_duration	`Float64`	`Float64`	`0.0`	Maximum ratio of installed energy to discharged capacity that can be installed (hours).
min_capacity_storage	`Float64`	`Float64`	`0.0`	Minimum allowed storage capacity (MWh).
min_duration	`Float64`	`Float64`	`0.0`	Minimum ratio of installed energy to discharged capacity that can be installed (hours).
min_outflow_fraction	`Float64`	`Float64`	`0.0`	Minimum outflow as a fraction of capacity.
min_storage_level	`Float64`	`Float64`	`0.0`	Minimum storage level as a fraction of capacity.
max_storage_level	`Float64`	`Float64`	`1.0`	Maximum storage level as a fraction of capacity.
storage_loss_fraction	`Float64`	Number $\in$ [0,1]	`0.0`	Fraction of stored commodity lost per timestep.

Attribute	Type	Values	Default	Description
type	`String`	`Electricity`	Required	Commodity of the edge.
start_vertex	`String`	Any electricity node id present in the system	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any electricity node id present in the system	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number $\in [0,1]$	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number $\in [0,1]$	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
ramp_up_fraction	`Float64`	Number $\in [0,1]$	`1.0`	Maximum increase in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
variable_om_cost	`Float64`	`Float64`	`0.0`	Variable operation and maintenance cost (USD/MWh).

Attribute	**Asset**	**Type**	**Values**	**Default**	**Description**
timedata	All	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "Electricity".
constraints	All	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.

Attribute	Type	Values	Default	Description
end_vertex	`String`	Any electricity node id present in the system	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"MustRunConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number $\in$ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number $\in$ [0,1]	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
ramp_up_fraction	`Float64`	Number $\in$ [0,1]	`1.0`	Maximum increase in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
unidirectional	`Bool`	`Bool`	`true`	Whether the edge is unidirectional.
variable_om_cost	`Float64`	`Float64`	`0.0`	Variable operation and maintenance cost (USD/MWh).

Attribute	Type	Values	Default	Description/Units
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "NaturalGas".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
capture_rate $\epsilon_{co2\_capture\_rate}$	`Float64`	`Float64`	1.0	t_{CO2}/MWh_{ng}
electricity_production ϵ_{elec_prod}	`Float64`	`Float64`	0.0	MWh_{elec}/MWh_{ng}
emission_rate $\epsilon_{emission\_rate}$	`Float64`	`Float64`	1.0	t_{CO2}/MWh_{ng}
fuel_consumption $\epsilon_{fuel\_consumption}$	`Float64`	`Float64`	1.0	MWh_{ng}/t_{CO2}

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "natgas_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints**

Attribute	Type	Values	Default	Description
type	`String`	`Electricity`	Required	Commodity flowing through the edge.
start_vertex	`String`	Any electricity node id present in the system	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any electricity node id present in the system	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
distance	`Float64`	`Float64`	`0.0`	Distance between the start and end vertex of the edge.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
ramp_up_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum increase in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
variable_om_cost	`Float64`	`Float64`	`0.0`	Variable operation and maintenance cost (USD/MWh).

Attribute	Type	Values	Default	Description/Units
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "NaturalGas".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
electricity_consumption $\epsilon_{elec\_consumption}$	`Float64`	`Float64`	0.0	MWh_{elec}/MWh_{h2}
efficiency_rate $\epsilon_{efficiency}$	`Float64`	`Float64`	1.0	MWh_{h2}/MWh_{fuel}
emission_rate $\epsilon_{emission\_rate}$	`Float64`	`Float64`	1.0	t_{CO2}/MWh_{fuel}
capture_rate $\epsilon_{co2\_capture\_rate}$	`Float64`	`Float64`	1.0	t_{CO2}/MWh_{fuel}

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Hydrogen".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge **(only available for the `Hydrogen` and `Fuel` edges)**.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
min_down_time	`Int64`	`Int64`	`0`	Minimum amount of time the edge has to remain in

Attribute	Type	Values	Default	Description/Units
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "NaturalGas".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.
efficiency_rate $\epsilon_{efficiency}$	`Float64`	`Float64`	1.0	MWh_{elec}/MWh_{fuel}
emission_rate $\epsilon_{emission\_rate}$	`Float64`	`Float64`	1.0	t_{CO2}/MWh_{fuel}
capture_rate $\epsilon_{co2\_capture\_rate}$	`Float64`	`Float64`	1.0	t_{CO2}/MWh_{fuel}

Attribute	Type	Values	Default	Description
type	`String`	Any MACRO commodity type matching the commodity of the edge	Required	Commodity of the edge. E.g. "Electricity".
start_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the starting vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
end_vertex	`String`	Any node id present in the system matching the commodity of the edge	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_2".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"CapacityConstraint": true}`.
availability	`Dict`	Availability file path and header	Empty	Path to the availability file and column name for the availability time series to link to the edge. E.g. `{"timeseries": {"path": "system/availability.csv", "header": "Availability_MW_z1"}}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge **(only available for the `Hydrogen` and `Fuel` edges)**.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
loss_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Fraction of transmission loss.
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
min_down_time	`Int64`	`Int64`	`0`	Minimum amount of time the edge has to remain in

Attribute	Type	Values	Default	Description
timedata	`String`	`String`	Required	Time resolution for the time series data linked to the transformation. E.g. "Electricity".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for vertices	Empty	List of constraints applied to the transformation. E.g. `{"BalanceConstraint": true}`.

Attribute	Type	Values	Default	Description
type	`String`	`Electricity`	Required	Commodity of the edge.
end_vertex	`String`	Any electricity node id present in the system	Required	ID of the ending vertex of the edge. The node must be present in the `nodes.json` file. E.g. "elec_node_1".
constraints	`Dict{String,Bool}`	Any MACRO constraint type for Edges	Empty	List of constraints applied to the edge. E.g. `{"MustRunConstraint": true}`.
can_expand	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity expansion.
can_retire	`Bool`	`Bool`	`false`	Whether the edge is eligible for capacity retirement.
capacity_size	`Float64`	`Float64`	`1.0`	Size of the edge capacity.
existing_capacity	`Float64`	`Float64`	`0.0`	Existing capacity of the edge in MW.
fixed_om_cost	`Float64`	`Float64`	`0.0`	Fixed operations and maintenance cost (USD/MW-year).
has_capacity	`Bool`	`Bool`	`false`	Whether capacity variables are created for the edge.
integer_decisions	`Bool`	`Bool`	`false`	Whether capacity variables are integers.
investment_cost	`Float64`	`Float64`	`0.0`	Annualized capacity investment cost (USD/MW-year)
max_capacity	`Float64`	`Float64`	`Inf`	Maximum allowed capacity of the edge (MW). **Note: add the `MaxCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_capacity	`Float64`	`Float64`	`0.0`	Minimum allowed capacity of the edge (MW). **Note: add the `MinCapacityConstraint` to the constraints dictionary to activate this constraint**.
min_flow_fraction	`Float64`	Number ∈ [0,1]	`0.0`	Minimum flow of the edge as a fraction of the total capacity. **Note: add the `MinFlowConstraint` to the constraints dictionary to activate this constraint**.
ramp_down_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum decrease in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
ramp_up_fraction	`Float64`	Number ∈ [0,1]	`1.0`	Maximum increase in flow between two time steps, reported as a fraction of the capacity. **Note: add the `RampingLimitConstraint` to the constraints dictionary to activate this constraint**.
unidirectional	`Bool`	`Bool`	`true`	Whether the edge is unidirectional.
variable_om_cost	`Float64`	`Float64`	`0.0`	Variable operation and maintenance cost (USD/MWh).

Chapter 11

Constraints

11.1 MACRO Constraint Library

Currently, Macro includes the following constraints:

Balance constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::BalanceConstraint, v::AbstractVertex, model::Model)
```

Add a balance constraint to the vertex `v`.

- If `v` is a `Node`, a demand balance constraint is added.
- If `v` is a `Transformation`, this constraint ensures that the stoichiometric equations linking the input and output flows are correctly balanced.

$$\sum_{\substack{i \in \text{balance\_eqs\_ids(v)},\\ t \in \text{time\_interval(v)}}} \text{balance\_eq(v, i, t)} = 0.0$$

source

Capacity constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::CapacityConstraint, e::Edge, model::Model)
```

Add a capacity constraint to the edge `e`. If the edge is unidirectional, the functional form of the constraint is:

$$\text{flow(e, t)} \leq \text{availability(e, t)} \times \text{capacity(e)}$$

If the edge is bidirectional, the constraint is:

$$i \times \text{flow(e, t)} \leq \text{availability(e, t)} \times \text{capacity(e)}$$

for each time `t` in `time_interval(e)` for the edge `e` and each `i` in `[-1, 1]`. The function `availability` returns the time series of the capacity factor of the edge at time `t`.

source

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::CapacityConstraint, e::EdgeWithUC, model::Model)
```

Add a capacity constraint to the edge e with unit commitment. If the edge is unidirectional, the functional form of the constraint is:

$$\sum_{t \in \text{time\_interval(e)}} \text{flow(e, t)} \leq \text{availability(e, t)} \times \text{capacity(e)} \times \text{ucommit(e, t)}$$

If the edge is bidirectional, the constraint is:

$$i \times \text{flow(e, t)} \leq \text{availability(e, t)} \times \text{capacity(e)} \times \text{ucommit(e, t)}$$

for each time `t` in `time_interval(e)` for the edge `e` and each `i` in `[-1, 1]`. The function `availability` returns the time series of the availability of the edge at time `t`.

source

CO2 capture constraint

Maximum capacity constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MaxCapacityConstraint, e::Edge, model::Model)
```

Add a max capacity constraint to the edge `e`. The functional form of the constraint is:

$$\text{capacity(e)} \leq \text{max\_capacity(e)}$$

source

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MaxCapacityConstraint, s::Storage, model::Model)
```

Add a max capacity constraint to the storage `s`. The functional form of the constraint is:

$$\text{capacity\_storage(s)} \leq \text{max\_capacity\_storage(s)}$$

source

Maximum non-served demand constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MaxNonServedDemandConstraint, n::Node, model::Model)
```

Add a max non-served demand constraint to the node `n`. The functional form of the constraint is:

$$\sum_{s\ \in\ \text{segments\_non\_served\_demand(n)}} \text{non\_served\_demand(n, s, t)} \leq \text{demand(n, t)}$$

for each time `t` in `time_interval(n)` for the node `n`.

source

Maximum non-served demand per segment constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(
    ct::MaxNonServedDemandPerSegmentConstraint,
    n::Node,
    model::Model,
)
```

Add a max non-served demand per segment constraint to the node `n`. The functional form of the constraint is:

$$\text{non\_served\_demand(n, s, t)} \leq \text{max\_non\_served\_demand(n, s)} \times \text{demand(n, t)}$$

for each segment `s` in `segments_non_served_demand(n)` and each time `t` in `time_interval(n)` for the node `n`. The function `segments_non_served_demand` returns the segments of the non-served demand of the node `n` as defined in the input data `nodes.json`.

source

Maximum storage level constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MaxStorageLevelConstraint, s::Storage, model::Model)
```

Add a max storage level constraint to the storage g. The functional form of the constraint is:

$$\text{storage\_level(s, t)} \leq \text{max\_storage\_level(s)} \times \text{capacity\_storage(s)}$$

for each time `t` in `time_interval(s)` for the storage `s`.

source

Minimum capacity constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinCapacityConstraint, e::Edge, model::Model)
```

Add a min capacity constraint to the edge `e`. The functional form of the constraint is:

$$\text{capacity(e)} \geq \text{min\_capacity(e)}$$

source

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinCapacityConstraint, s::Storage, model::Model)
```

Add a min capacity constraint to the storage `s`. The functional form of the constraint is:

$$\text{capacity\_storage(s)} \geq \text{min\_capacity\_storage(s)}$$

source

Minimum flow constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinFlowConstraint, e::Edge, model::Model)
```

Add a min flow constraint to the edge `e`. The functional form of the constraint is:

$$\text{flow(e, t)} \geq \text{min\_flow\_fraction(e)} \times \text{capacity(e)}$$

for each time `t` in `time_interval(e)` for the edge `e`.

Note

This constraint is available only for unidirectional edges.

source

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinFlowConstraint, e::EdgeWithUC, model::Model)
```

Add a min flow constraint to the edge e with unit commitment. The functional form of the constraint is:

$$\text{flow(e, t)} \geq \text{min\_flow\_fraction(e)} \times \text{capacity\_size(e)} \times \text{ucommit(e, t)}$$

for each time `t` in `time_interval(e)` for the edge `e`.

Note

This constraint is available only for unidirectional edges.

source

Minimum storage level constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinStorageLevelConstraint, s::Storage, model::Model)
```

Add a min storage level constraint to the storage `s`. The functional form of the constraint is:

$$\text{storage\_level(s, t)} \geq \text{min\_storage\_level(s)} \times \text{capacity\_storage(s)}$$

for each time `t` in `time_interval(s)` for the storage `s`.

source

Minimum storage outflow constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinStorageOutflowConstraint, s::Storage, model::Model)
```

Add a min storage outflow constraint to the storage `s` part of a HydroRes asset. The functional form of the constraint is:

$$\text{flow(spillage\_edge, t)} + \text{flow(discharge\_edge, t)} \geq \text{min\_outflow\_fraction(s)} \times \text{capacity(discharge\_edge)}$$

for each time `t` in `time_interval(s)` for the storage `s`.

Only applies to HydroRes assets

This constraint only applies to HydroRes assets. It returns a warning if the storage `s` does not have a spillage edge. If the discharge edge is the only outflow, you should apply MinFlowConstraint to the discharge edge.

source

Minimum up and down time constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinUpTimeConstraint, e::EdgeWithUC, model::Model)
```

Add a min up time constraint to the edge e with unit commitment. The functional form of the constraint is:

$$\text{ucommit(e, t)} \geq \sum_{h=0}^{\text{min\_up\_time(e)}-1} \text{ustart(e, t-h)}$$

for each time `t` in `time_interval(e)` for the edge `e`. The function `timestepbefore` is used to perform the time wrapping within the subperiods and get the correct time step before `t`.

Min up time duration

This constraint will throw an error if the minimum up time is longer than the length of one subperiod.

source

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MinDownTimeConstraint, e::EdgeWithUC, model::Model)
```

Add a min down time constraint to the edge e with unit commitment. The functional form of the constraint is:

$$\frac{\text{capacity(e)}}{\text{capacity\_size(e)}} - \text{ucommit(e, t)} \geq \sum_{h=0}^{\text{min\_down\_time(e)}-1} \text{ushut(e, t-h)}$$

for each time `t` in `time_interval(e)` for the edge `e`. The function `timestepbefore` is used to perform the time wrapping within the subperiods and get the correct time step before `t`.

Min down time duration

This constraint will throw an error if the minimum down time is longer than the length of one subperiod.

source

Must-run constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::MustRunConstraint, e::Edge, model::Model)
```

Add a must run constraint to the edge `e`. The functional form of the constraint is:

$$\text{flow(e, t)} = \text{availability(e, t)} \times \text{capacity(e)}$$

for each time `t` in `time_interval(e)` for the edge `e`.

Must run constraint

This constraint is available only for unidirectional edges.

source

Ramping limits constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::RampingLimitConstraint, e::Edge, model::Model)
```

Add a ramping limit constraint to the edge `e`. The functional form of the ramping up limit constraint is:

$$\text{flow(e, t)} - \text{flow(e, t-1)} + \text{regulation\_term(e, t)} + \text{reserves\_term(e, t)} - \text{ramp\_up\_fraction(e)} \times \text{capacity(e)} \leq 0$$

On the other hand, the ramping down limit constraint is:

$$\text{flow(e, t-1)} - \text{flow(e, t)} + \text{regulation\_term(e, t)} + \text{reserves\_term(e, t)} - \text{ramp\_down\_fraction(e)} \times \text{capacity(e)} \leq 0$$

for each time `t` in `time_interval(e)` for the edge `e`. The function `timestepbefore` is used to perform the time wrapping within the subperiods and get the correct time step before `t`.

source

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::RampingLimitConstraint, e::EdgeWithUC, model::Model)
```

Add a ramping limit constraint to the edge e with unit commitment. The functional form of the ramping up limit constraint is:

$$\text{flow(e, t)} - \text{flow(e, t-1)} + \text{regulation\_term(e, t)} + \text{reserves\_term(e, t)} - \text{ramp\_up\_fraction(e)} \times \text{capacity\_size(e)} \times (\text{ucommit}$$

On the other hand, the ramping down limit constraint is:

$$\text{flow(e, t-1)} - \text{flow(e, t)} + \text{regulation\_term(e, t)} + \text{reserves\_term(e, t)} - \text{ramp\_down\_fraction(e)} \times \text{capacity\_size(e)} \times (\text{ucom}$$

for each time `t` in `time_interval(e)` for the edge e. The function `timestepbefore` is used to perform the time wrapping within the subperiods and get the correct time step before `t`.

source

Storage capacity constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::StorageCapacityConstraint, s::Storage, model::Model)
```

Add a storage capacity constraint to the storage `s`. The functional form of the constraint is:

$$\text{storage\_level(s, t)} \leq \text{capacity\_storage(s)}$$

for each time `t` in `time_interval(s)` for the storage `s`.

source

Storage discharge limit constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::StorageDischargeLimitConstraint, e::Edge, model::Model)
```

Add a storage discharge limit constraint to the edge `e` if the start vertex of the edge is a storage. The functional form of the constraint is:

$$\frac{\text{flow(e, t)}}{\text{efficiency(e)}} \leq \text{storage\_level(start\_vertex(e), timestepbefore(t, 1, subperiods(e)))}$$

for each time `t` in `time_interval(e)` for the edge `e`. The function `timestepbefore` is used to perform the time wrapping within the subperiods and get the correct time step before `t`.

Storage discharge limit constraint

This constraint is only applied to edges with a start vertex that is a storage.

source

Storage symmetric capacity constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(
    ct::StorageSymmetricCapacityConstraint,
    s::Storage,
    model::Model,
)
```

Add a storage symmetric capacity constraint to the storage `s`. The functional form of the constraint is:

$$\text{flow(e\_discharge, t)} + \text{flow(e\_charge, t)} \leq \text{capacity(e\_discharge)}$$

source

Storage charge discharge ratio constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(
    ct::StorageChargeDischargeRatioConstraint,
    s::Storage,
    model::Model,
)
```

Add a storage charge discharge ratio constraint to the storage `s`. The functional form of the constraint is:

$$\text{charge\_discharge\_ratio(s)} \times \text{capacity(s.discharge\_edge)} = \text{capacity(s.charge\_edge)}$$

source

Storage max duration constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::StorageMaxDurationConstraint, s::Storage, model::Model)
```

Add a storage max duration constraint to the storage `s`. The functional form of the constraint is:

$$\text{capacity\_storage(s)} \leq \text{max\_duration(s)} \times \text{capacity(discharge\_edge(s))}$$

Storage max duration constraint

This constraint is only applied if the maximum duration is greater than 0.

source

Storage min duration constraint

`Macro.add_model_constraint!` – Method.

```
add_model_constraint!(ct::StorageMinDurationConstraint, s::Storage, model::Model)
```

Add a storage min duration constraint to the storage `s`. The functional form of the constraint is:

$$\text{capacity\_storage(s)} \geq \text{min\_duration(s)} \times \text{capacity(discharge\_edge(s))}$$

Storage min duration constraint

This constraint is only applied if the minimum duration is greater than 0.

source

Part V

Modeler Guide

Chapter 12

How to build a sector

12.1 Modeler Guide

How to build new sectors in MACRO

Overview

The steps to build a sector in MACRO are as follows:

1. Create new sectors/commodity types by defining new subtypes of `Commodity` in the `Macro.jl` file.
2. Create new assets. Each asset type should be a subtype of `AbstractAsset` and be defined in a Julia (`.jl`) file located in the `src/assets` folder. A `make` function should be defined for each asset type to create an instance of the asset.

Note

Remember to include the new files in the `Macro.jl` file, so that they are available when the package is loaded.

During the creation of the assets, you will need to provide (check the following sections for an example):

1. **Asset structure**: list of fields that define the asset in the form of transformations, edges, and storage units.
2. **Default constraints** for the transformations, edges, and storage units.
3. **Stoichiometric equations/coefficients** for the transformation processes.

The following section provides an example of how to create a new sector and assets in MACRO.

Example

For example, let's create a new sector called `MyNewSector` with two assets: `MyAsset1`, and `MyAsset2`.

The first asset will be a technology that converts a commodity `MyNewSector`, into two other commodities, `Electricity` and `CO2`, while the second asset will be a technology with a `storage unit` that stores the commodity `MyNewSector`.

As seen in the previous section, the steps to create a new sector and assets are as follows:

- Add the following line to the Macro.jl file:

```
abstract type MyNewSector <: Commodity end
```

(try to add this line right after the definition of the `Commodity` type).

- Create a new file called `MyAsset1.jl` in the `src/assets` folder with the following content:

```
# Structure of the asset
struct MyAsset1 <: AbstractAsset
    id::AssetId
    myasset1_transform::Transformation
    mynewsector_edge::Edge{MyNewSector}
    e_edge::Edge{Electricity}
    co2_edge::Edge{CO2}
end

# Make function to create an instance of the asset
# The function takes as input the data and the system, and returns an instance of the asset
# The data is a dictionary with the asset data, and the system is the system object containing the
↪  locations, time data, and other relevant information
function make(::Type{MyAsset1}, data::AbstractDict{Symbol,Any}, system::System)

    # asset id
    id = AssetId(data[:id])

    # transformation
    transform_data = process_data(data[:transforms])
    myasset1_transform_default_constraints = [BalanceConstraint()]
    myasset1_transform = Transformation(;
        id = Symbol(transform_data[:id]),
        timedata = system.time_data[Symbol(transform_data[:timedata])],
        constraints = get(transform_data, :constraints, myasset1_transform_default_constraints),
    )

    # edges
    mynewsector_edge_data = process_data(data[:edges][:mynewsector_edge])
    mynewsector_edge_default_constraints = Vector{AbstractTypeConstraint}()
    mynewsector_start_node = find_node(system.locations,
    ↪  Symbol(mynewsector_edge_data[:start_vertex]))
    mynewsector_end_node = myasset1_transform
    mynewsector_edge = Edge(
        Symbol(String(id) * "_" * mynewsector_edge_data[:id]),
        mynewsector_edge_data,
        system.time_data[:MyNewSector],
        MyNewSector,
        mynewsector_start_node,
        mynewsector_end_node,
    )
    mynewsector_edge.constraints = get(mynewsector_edge_data, :constraints,
    ↪  mynewsector_edge_default_constraints)
    mynewsector_edge.unidirectional = get(mynewsector_edge_data, :unidirectional, true)
    elec_edge_data = process_data(data[:edges][:e_edge])
    elec_start_node = myasset1_transform
    elec_end_node = find_node(system.locations, Symbol(elec_edge_data[:end_vertex]))
    elec_edge = EdgeWithUC(
        Symbol(String(id) * "_" * elec_edge_data[:id]),
        elec_edge_data,
        system.time_data[:Electricity],
        Electricity,
        elec_start_node,
        elec_end_node,
    )
    elec_edge.constraints = get(
        elec_edge_data,
        :constraints,
        [
            CapacityConstraint(),
            RampingLimitConstraint(),
            MinUpTimeConstraint(),
            MinDownTimeConstraint(),
        ],
    )
    elec_edge.unidirectional = get(elec_edge_data, :unidirectional, true)
    elec_edge.startup_fuel_balance_id = :energy

    co2_edge_data = process_data(data[:edges][:co2_edge])
    co2_start_node = myasset1_transform
    co2_end_node = find_node(system.locations, Symbol(co2_edge_data[:end_vertex]))
    co2_edge = Edge(
        Symbol(String(id) * "_" * co2_edge_data[:id]),
        co2_edge_data,
        system.time_data[:CO2],
        CO2,
        co2_start_node,
        co2_end_node,
    )
    co2_edge.constraints =
        get(co2_edge_data, :constraints, Vector{AbstractTypeConstraint}())
    co2_edge.unidirectional = get(co2_edge_data, :unidirectional, true)

    myasset1_transform.balance_data = Dict(
        # Edit this part to include the stoichiometric equations for the transformation process
        ),
    )

    return MyAsset1(id, myasset1_transform, mynewsector_edge, elec_edge, co2_edge)
end
```

From the code above, you can see that the modeler needs to provide the asset structure as a Julia `struct`, along with the default constraints for transformations and edges (`myasset1_transform_default_constraints`, `mynewsector_edge_de` and the stoichiometric coefficients for the transformation process being modeled (`myasset1_transform.balance_data`).

Tip

Checking out other asset files in the `src/assets` folder is a good place to start adding new assets.

The creation of the second asset, `MyAsset2`, follows very similar steps to the creation of `MyAsset1`. The main difference is that `MyAsset2` has a storage unit:

```julia
struct MyAsset2 <: AbstractAsset
    id::AssetId
    myasset2_storage::Storage{MyNewSector}  # <--- Storage unit
    discharge_edge::Edge{MyNewSector}
    charge_edge::Edge{MyNewSector}
end

function make(::Type{MyAsset2}, data::AbstractDict{Symbol,Any}, system::System)

    # asset id
    id = AssetId(data[:id])

    # storage
    storage_data = process_data(data[:storage])
    myasset2_storage_default_constraints = [
            BalanceConstraint(),
            StorageCapacityConstraint(),
            StorageMaxDurationConstraint(),
            StorageMinDurationConstraint(),
            StorageSymmetricCapacityConstraint(),
        ]
    myasset2_storage = Storage(id,
        storage_data,
        system.time_data[Symbol(storage_data[:commodity])],
        MyNewSector,
        myasset2_storage_default_constraints
    )

    # edges
    discharge_edge_data = process_data(data[:edges][:discharge_edge])
    discharge_edge_default_constraints = [CapacityConstraint()]
    discharge_start_node = myasset2_storage
    discharge_end_node = find_node(system.locations, Symbol(discharge_edge_data[:end_vertex]))
    discharge_edge = Edge(
        Symbol(String(id) * "_" * discharge_edge_data[:id]),
        discharge_edge_data,
        system.time_data[:MyNewSector],
        MyNewSector,
        discharge_start_node,
        discharge_end_node,
    )
    discharge_edge.constraints = get(discharge_edge_data, :constraints,
    ↪  discharge_edge_default_constraints)
    discharge_edge.unidirectional = get(discharge_edge_data, :unidirectional, true)

    charge_edge_data = process_data(data[:edges][:charge_edge])
    charge_start_node = find_node(system.locations, Symbol(charge_edge_data[:start_vertex]))
    charge_end_node = myasset2_storage
    charge_edge = Edge(
        Symbol(String(id) * "_" * charge_edge_data[:id]),
        charge_edge_data,
        system.time_data[:MyNewSector],
        MyNewSector,
        charge_start_node,
        charge_end_node,
    )
    charge_edge.constraints = get(charge_edge_data, :constraints, Vector{AbstractTypeConstraint}())
    charge_edge.unidirectional = get(charge_edge_data, :unidirectional, true)

    myasset2_storage.discharge_edge = discharge_edge
    myasset2_storage.charge_edge = charge_edge

    myasset2_storage.balance_data = Dict(
        # Edit this part to include the energy efficiency of the storage unit or any other
        ↪  stoiometric equations
        ),
    )

    return MyAsset2(id, myasset2_storage, discharge_edge, charge_edge)
end
```

Chapter 13

How to create an example case

13.1 How to create an example case to test the new sectors and assets

Once new sectors and assets have been created in the model, you may want to test them by creating a new example case. This section explains how to achieve this.

The best way to create a new example case is to include the new sectors and assets in an existing example case. They can be found in the `ExampleSystems` folder in the `Macro` repository.

An example case is a folder that contains all the necessary data files to run the model. The case folder should have the following structure:

```
MyCase
│
├── ▯ settings
│   └── macro_settings.yml
│
├── ▯ system
│   ├── commodities.json
│   ├── time_data.json
│   ├── nodes.json
│   ├── fuel_prices.csv
│   └── demand.csv
│
├── ▯ assets
│   ├── MyAsset1.json
│   ├── MyAsset2.json
│ [...other asset types...]
│   └── availability.csv
│
└── system_data.json
```

When adding a new sector, you need to make sure that:

- The new sector is included in the `commodities.json` file.
- The new sector is included in the `time_data.json` file, with the corresponding `HoursPerTimeStep` and `HoursPerSubperiod` values.
- Nodes corresponding to the new sector are included in the `nodes.json` file.

- The demand corresponding to the new sector and for each node is included in the `demand.csv` file.
- A new JSON file is created with the data for the new assets.
- `Availability.csv` and `fuel_prices.csv` files are updated with the availability and fuel prices for the new assets (if applicable).

Warning

Make sure that the values of the `type` keys in the JSON files match the names of the new sector and assets (julia `abstract type` and `struct` names respectively) created in the model. The same applies to the keys in the `commodities.json` and `time_data.json` files.

Part VI

Developer Guide

Chapter 14

Type hierarchy

14.1 Developer Guide

MACRO type hierarchy

Commodities

```
print_tree(Commodity)
```

```
Commodity
├─ Biomass
├─ CO2
│  └─ CO2Captured
├─ Coal
├─ Electricity
├─ Hydrogen
├─ NaturalGas
└─ Uranium
```

Assets

```
print_tree(AbstractAsset)
```

```
AbstractAsset
├─ BECCSElectricity
├─ BECCSHydrogen
├─ Battery
├─ ElectricDAC
├─ Electrolyzer
├─ FuelCell
├─ GasStorage
├─ HydroRes
├─ HydrogenLine
├─ MustRun
├─ NaturalGasDAC
├─ PowerLine
├─ ThermalHydrogen
├─ ThermalHydrogenCCS
├─ ThermalPower
├─ ThermalPowerCCS
└─ VRE
```

Constraints

```
print_tree(Macro.AbstractTypeConstraint)
```

```
AbstractTypeConstraint
├─ OperationConstraint
│  ├─ BalanceConstraint
│  ├─ CapacityConstraint
│  ├─ MaxNonServedDemandConstraint
│  ├─ MaxNonServedDemandPerSegmentConstraint
│  ├─ MaxStorageLevelConstraint
│  ├─ MinDownTimeConstraint
│  ├─ MinFlowConstraint
│  ├─ MinStorageLevelConstraint
│  ├─ MinStorageOutflowConstraint
│  ├─ MinUpTimeConstraint
│  ├─ MustRunConstraint
│  ├─ PolicyConstraint
│  │  └─ CO2CapConstraint
│  ├─ RampingLimitConstraint
│  ├─ StorageCapacityConstraint
│  ├─ StorageDischargeLimitConstraint
│  └─ StorageSymmetricCapacityConstraint
└─ PlanningConstraint
   ├─ MaxCapacityConstraint
   ├─ MinCapacityConstraint
   ├─ StorageChargeDischargeRatioConstraint
   ├─ StorageMaxDurationConstraint
   └─ StorageMinDurationConstraint
```

Part VII

References

Chapter 15

Macro Objects

15.1 MACRO Objects

Vertex

`Macro.@AbstractVertexBaseAttributes` – Macro.

```
@AbstractVertexBaseAttributes()

A macro that defines the base attributes for all vertex types in the network model.

# Generated Fields
- id::Symbol: Unique identifier for the vertex
- timedata::TimeData: Time-related data for the vertex
- balance_data::Dict{Symbol,Dict{Symbol,Float64}}: Dictionary mapping balance equation IDs to
↪  coefficients
- constraints::Vector{AbstractTypeConstraint}: List of constraints applied to the vertex
- operation_expr::Dict: Dictionary storing operational JuMP expressions for the vertex

This macro is used to ensure consistent base attributes across all vertex types in the network.
```

source

Node

`Macro.Node` – Type.

```
Node{T} <: AbstractVertex

A mutable struct representing a node in a network, parameterized by commodity type T.

# Inherited Attributes
- id::Symbol: Unique identifier for the node
- timedata::TimeData: Time-related data for the node
- balance_data::Dict{Symbol,Dict{Symbol,Float64}}: Balance equations data
- constraints::Vector{AbstractTypeConstraint}: List of constraints applied to the node
- operation_expr::Dict: Operational JuMP expressions for the node
```

```
# Fields
- demand::Union{Vector{Float64},Dict{Int64,Float64}}: Time series of demand values
- max_nsd::Vector{Float64}: Maximum allowed non-served demand for each segment
- max_supply::Vector{Float64}: Maximum supply for each segment
- non_served_demand::Union{JuMPVariable,Matrix{Float64}}: JuMP variables or matrix representing
↪  unmet demand
- policy_budgeting_vars::Dict: Policy budgeting variables for constraints
- policy_slack_vars::Dict: Policy slack variables for constraints
- price::Union{Vector{Float64},Dict{Int64,Float64}}: Time series of prices
- price_nsd::Vector{Float64}: Penalties for non-served demand by segment
- price_supply::Vector{Float64}: Supply costs by segment
- price_unmet_policy::Dict{DataType,Float64}: Mapping of policy types to penalty costs
- rhs_policy::Dict{DataType,Float64}: Mapping of policy types to right-hand side values
- supply_flow::Union{JuMPVariable,Matrix{Float64}}: JuMP variables or matrix representing supply
↪  flows

Note: Base attributes are inherited from AbstractVertex via @AbstractVertexBaseAttributes macro.
```

source

Transformation

`Macro.Transformation` – Type.

```
Transformation <: AbstractVertex

A mutable struct representing a transformation vertex in a network model, which models a
↪  conversion process between different commodities or energy forms.

# Inherited Attributes
- id::Symbol: Unique identifier for the transformation
- timedata::TimeData: Time-related data for the transformation
- balance_data::Dict{Symbol,Dict{Symbol,Float64}}: Dictionary mapping stoichiometric equation
↪  IDs to coefficients
- constraints::Vector{AbstractTypeConstraint}: List of constraints applied to the transformation
- operation_expr::Dict: Dictionary storing operational JuMP expressions for the transformation

Transformations are used to model conversion processes between different commodities, such as
↪  power plants
converting fuel to electricity or electrolyzers converting electricity to hydrogen. The
↪  `balance_data` field
typically contains conversion efficiencies and other relationships between input and output
↪  flows.
```

source

Storage

`Macro.Storage` – Type.

```
Storage{T} <: AbstractVertex

A mutable struct representing a storage vertex in a network model, parameterized by commodity
↪  type T.

# Inherited Attributes
- id::Symbol: Unique identifier for the storage
- timedata::TimeData: Time-related data for the storage
- balance_data::Dict{Symbol,Dict{Symbol,Float64}}: Dictionary mapping balance equation IDs to
↪  coefficients
- constraints::Vector{AbstractTypeConstraint}: List of constraints applied to the storage
- operation_expr::Dict: Dictionary storing operational JuMP expressions for the storage

# Fields
- can_expand::Bool: Whether storage capacity can be expanded
- can_retire::Bool: Whether storage capacity can be retired
- capacity_storage::Union{AffExpr,Float64}: Total available storage capacity
- charge_edge::Union{Nothing,AbstractEdge}: `Edge` representing charging flow
- charge_discharge_ratio::Float64: Ratio between charging and discharging rates
- discharge_edge::Union{Nothing,AbstractEdge}: `Edge` representing discharging flow
- existing_capacity_storage::Float64: Initial installed storage capacity
- fixed_om_cost_storage::Float64: Fixed operation and maintenance costs
- investment_cost_storage::Float64: Cost per unit of new storage capacity
- max_capacity_storage::Float64: Maximum allowed storage capacity
- max_duration::Float64: Maximum storage duration in hours
- min_capacity_storage::Float64: Minimum required storage capacity
- min_duration::Float64: Minimum storage duration in hours
- min_outflow_fraction::Float64: Minimum discharge rate as fraction of capacity
- min_storage_level::Float64: Minimum storage level as fraction of capacity
- max_storage_level::Float64: Maximum storage level as fraction of capacity
- new_capacity_storage::Union{JuMPVariable,Float64}: New storage capacity to be built
- ret_capacity_storage::Union{JuMPVariable,Float64}: Storage capacity to be retired
- spillage_edge::Union{Nothing,AbstractEdge}: Edge representing spillage/losses (e.g. hydro
↪  reservoirs)
- storage_level::Union{JuMPVariable,Vector{Float64}}: Storage level at each timestep
- storage_loss_fraction::Float64: Fraction of stored commodity lost per timestep

Storage vertices represent facilities that can store commodities over time, such as batteries,
pumped hydro, or gas storage. They can charge (store) and discharge (release) commodities,
subject to capacity and operational constraints.
```

source

Edge (with and without UC)

Macro.Edge – Type.

```
Edge{T} <: AbstractEdge{T}

A mutable struct representing an edge in a network model, parameterized by commodity type T.

# Fields
- id::Symbol: Unique identifier for the edge
- timedata::TimeData: Time-related data for the edge
- start_vertex::AbstractVertex: Starting vertex of the edge
- end_vertex::AbstractVertex: Ending vertex of the edge
- availability::Vector{Float64}: Time series of availability factors
- can_expand::Bool: Whether edge capacity can be expanded
- can_retire::Bool: Whether edge capacity can be retired
- capacity::Union{AffExpr,Float64}: Total available capacity
- capacity_size::Float64: Size factor for resource cluster
- constraints::Vector{AbstractTypeConstraint}: List of constraints applied to the edge
- distance::Float64: Physical distance of the edge
- existing_capacity::Float64: Initial installed capacity
- fixed_om_cost::Float64: Fixed operation and maintenance costs
- flow::Union{JuMPVariable,Vector{Float64}}: Flow of commodity `T` through the edge at each
↪  timestep
- has_capacity::Bool: Whether the edge has capacity variables
- integer_decisions::Bool: Whether capacity decisions must be integer
- investment_cost::Float64: Cost per unit of new capacity
- loss_fraction::Float64: Fraction of flow lost during transmission
- max_capacity::Float64: Maximum allowed capacity
- min_capacity::Float64: Minimum required capacity
- min_flow_fraction::Float64: Minimum flow as fraction of capacity
- new_capacity::Union{JuMPVariable,Float64}: JuMP variable representing new capacity built
- ramp_down_fraction::Float64: Maximum ramp-down rate as fraction of capacity
- ramp_up_fraction::Float64: Maximum ramp-up rate as fraction of capacity
- ret_capacity::Union{JuMPVariable,Float64}: JuMP variable representing capacity to be retired
- unidirectional::Bool: Whether flow is restricted to one direction
- variable_om_cost::Float64: Variable operation and maintenance costs per unit flow

Edges represent connections between vertices that allow commodities to flow between them.
They can model physical infrastructure like pipelines, transmission lines, or logical
connections with associated costs, capacities, and operational constraints.
```

source

Macro.EdgeWithUC – Type.

```
EdgeWithUC{T} <: AbstractEdge{T}

A mutable struct representing an edge with unit commitment constraints in a network model,
↪  parameterized by commodity type T.

# Inherited Attributes from Edge
- id::Symbol: Unique identifier for the edge
- timedata::TimeData: Time-related data for the edge
- start_vertex::AbstractVertex: Starting vertex of the edge
- end_vertex::AbstractVertex: Ending vertex of the edge
- availability::Vector{Float64}: Time series of availability factors
- can_expand::Bool: Whether edge capacity can be expanded
- can_retire::Bool: Whether edge capacity can be retired
- capacity::Union{AffExpr,Float64}: Total available capacity
- capacity_size::Float64: Size factor for resource cluster
- constraints::Vector{AbstractTypeConstraint}: List of constraints applied to the edge
- distance::Float64: Physical distance of the edge
- existing_capacity::Float64: Initial installed capacity
- fixed_om_cost::Float64: Fixed operation and maintenance costs
- flow::Union{JuMPVariable,Vector{Float64}}: Flow of commodity through the edge at each timestep
- has_capacity::Bool: Whether the edge has capacity variables
- integer_decisions::Bool: Whether capacity decisions must be integer
- investment_cost::Float64: Cost per unit of new capacity
- loss_fraction::Float64: Fraction of flow lost during transmission
- max_capacity::Float64: Maximum allowed capacity
- min_capacity::Float64: Minimum required capacity
- min_flow_fraction::Float64: Minimum flow as fraction of capacity
- new_capacity::Union{JuMPVariable,Float64}: JuMP variable representing new capacity built
- ramp_down_fraction::Float64: Maximum ramp-down rate as fraction of capacity
- ramp_up_fraction::Float64: Maximum ramp-up rate as fraction of capacity
- ret_capacity::Union{JuMPVariable,Float64}: JuMP variable representing capacity to be retired
- unidirectional::Bool: Whether flow is restricted to one direction
- variable_om_cost::Float64: Variable operation and maintenance costs per unit flow

# Fields specific to EdgeWithUC
- min_down_time::Int64: Minimum time units that must elapse between shutting down and starting
↪  up
- min_up_time::Int64: Minimum time units that must elapse between starting up and shutting down
- startup_cost::Float64: Cost incurred when starting up the unit
- startup_fuel::Float64: Amount of fuel consumed during startup
- startup_fuel_balance_id::Symbol: Identifier for the balance constraint tracking startup fuel
- ucommit::Union{JuMPVariable,Vector{Float64}}: Binary commitment state variables
- ushut::Union{JuMPVariable,Vector{Float64}}: Binary shutdown decision variables
- ustart::Union{JuMPVariable,Vector{Float64}}: Binary startup decision variables

EdgeWithUC extends Edge to model units that have operational constraints related to their on/off
↪  status. It includes variables and parameters
for tracking unit commitment decisions and associated costs/constraints.
```

source

Chapter 16

Asset Library

16.1 MACRO Asset Library Reference

VRE (make function)

`Macro.make` – Method.

```
make(::Type{<:VRE}, data::AbstractDict{Symbol, Any}, system::System) -> VRE

VRE is an alias for Union{SolarPV, WindTurbine}

Necessary data fields:
 - transforms: Dict{Symbol, Any}
    - id: String
    - timedata: String
- edges: Dict{Symbol, Any}
    - edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
```

source

Battery (make function)

`Macro.make` – Method.

```
make(::Type{Battery}, data::AbstractDict{Symbol, Any}, system::System) -> Battery

Necessary data fields:
 - storage: Dict{Symbol, Any}
    - id: String
    - commodity: String
    - can_retire: Bool
    - can_expand: Bool
    - existing_capacity_storage: Float64
    - investment_cost_storage: Float64
    - fixed_om_cost_storage: Float64
    - storage_loss_fraction: Float64
    - min_duration: Float64
    - max_duration: Float64
    - min_storage_level: Float64
    - min_capacity_storage: Float64
    - max_capacity_storage: Float64
    - constraints: Vector{AbstractTypeConstraint}
 - edges: Dict{Symbol, Any}
    - charge_edge: Dict{Symbol, Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - efficiency: Float64
    - discharge_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - efficiency
        - constraints: Vector{AbstractTypeConstraint}
```

source

Electrolyzer (make function)

`Macro.make` – Method.

```
make(::Type{Electrolyzer}, data::AbstractDict{Symbol, Any}, system::System) -> Electrolyzer

Necessary data fields:
 - transforms: Dict{Symbol, Any}
    - id: String
    - timedata: String
    - efficiency_rate: Float64
    - constraints: Vector{AbstractTypeConstraint}
- edges: Dict{Symbol, Any}
    - h2_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
    - e_edge: Dict{Symbol, Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
```

source

ThermalHydrogen (make function)

`Macro.make` – Method.

```
make(::Type{ThermalHydrogen}, data::AbstractDict{Symbol, Any}, system::System) ->
↪  ThermalHydrogen

Necessary data fields:
 - transforms: Dict{Symbol, Any}
    - id: String
    - timedata: String
    - efficiency_rate: Float64
    - emission_rate: Float64
    - constraints: Vector{AbstractTypeConstraint}
- edges: Dict{Symbol, Any}
    - elec_edge: Dict{Symbol,Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
    - h2_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - min_up_time: Int
        - min_down_time: Int
        - startup_cost: Float64
        - startup_fuel: Float64
        - startup_fuel_balance_id: Symbol
        - constraints: Vector{AbstractTypeConstraint}
    - fuel_edge: Dict{Symbol, Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
    - co2_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
```

source

ThermalHydrogenCCS (make function)

`Macro.make` – Method.

```
make(::Type{ThermalHydrogenCCS}, data::AbstractDict{Symbol, Any}, system::System) ->
↪  ThermalHydrogenCCS

Necessary data fields:
 - transforms: Dict{Symbol, Any}
    - id: String
    - timedata: String
    - efficiency_rate: Float64
    - emission_rate: Float64
    - constraints: Vector{AbstractTypeConstraint}
- edges: Dict{Symbol, Any}
    - elec_edge: Dict{Symbol,Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
    - h2_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - min_up_time: Int
        - min_down_time: Int
        - startup_cost: Float64
        - startup_fuel: Float64
        - startup_fuel_balance_id: Symbol
        - constraints: Vector{AbstractTypeConstraint}
    - fuel_edge: Dict{Symbol, Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
    - co2_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
    - co2_captured_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
```

source

FuelCell (make function)

`Macro.make` – Method.

```
make(::Type{FuelCell}, data::AbstractDict{Symbol, Any}, system::System) -> FuelCell

Necessary data fields:
 - transforms: Dict{Symbol, Any}
    - id: String
    - timedata: String
    - efficiency_rate: Float64
    - constraints: Vector{AbstractTypeConstraint}
- edges: Dict{Symbol, Any}
    - h2_edge: Dict{Symbol, Any}
        - id: String
        - end_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
    - e_edge: Dict{Symbol, Any}
        - id: String
        - start_vertex: String
        - unidirectional: Bool
        - has_capacity: Bool
        - can_retire: Bool
        - can_expand: Bool
        - constraints: Vector{AbstractTypeConstraint}
```

source

Chapter 17

Utilities

17.1 Utilities

`Macro.get_value_and_keys` – Function.

```
get_value_and_keys(dict::AbstractDict, target_key::Symbol, keys=Symbol[])
```

Recursively searches for a target key in a dictionary and returns a list of tuples containing the value associated with the target key and the keys leading to it. This function is used to replace the path to a timeseries file with the actual vector of data.

Arguments

- `dict::AbstractDict`: The (nested) dictionary to search in.
- `target_key::Symbol`: The key to search for.
- `keys=Symbol[]`: (optional) The keys leading to the current dictionary.

Returns

- `value_keys`: A list of tuples, where each tuple contains - the value associated with the target key - the keys leading to it in the nested dictionary.

Examples

```
dict = Dict(:a => Dict(:b => 1, :c => Dict(:b => 2)))
get_value_and_keys(dict, :b) # returns [(1, [:a, :b]), (2, [:a, :c, :b])]
```

Where the first element of the tuple is the value of the key :b and the second element is the list of keys to reach that value.

source

`Macro.set_value` – Function.

```
set_value(dict::AbstractDict, keys::Vector{Symbol}, new_value)
```

Set the value of a nested dictionary given a list of keys.

Arguments

- `dict::AbstractDict`: The dictionary to modify.
- `keys::Vector{Symbol}`: A list of keys representing the path to the value to

be modified.

- `new_value`: The new value to set.

Examples

```
dict = Dict(:a => Dict(:b => 1, :c => Dict(:b => 2)))
set_value(dict, [:a, :b], 3)
get_value(dict, [:a, :b]) # returns 3
```

source

`Macro.timestepbefore` – Function.

```
timestepbefore(t::Int, h::Int,subperiods::Vector{StepRange{Int64,Int64}})
```

Determines the time step that is h steps before index `t` in subperiod p with circular indexing.

source

`Macro.struct_info` – Function.

```
struct_info(t::Type{T}) where T

Return a vector of tuples with the field names and types of a struct.
```

source

`Macro.get_value` – Function.

```
get_value(dict::AbstractDict, keys::Vector{Symbol})
```

Get the value from a dictionary based on a sequence of keys.

Arguments

- `dict::AbstractDict`: The dictionary from which to retrieve the value.
- `keys::Vector{Symbol}`: The sequence of keys to traverse the dictionary.

Returns

- The value retrieved from the dictionary based on the given keys.

Examples

```
dict = Dict(:a => Dict(:b => 1, :c => Dict(:b => 2)))
get_value(dict, [:a, :b]) # returns 1
get_value(dict, [:a, :c, :b]) # returns 2
```

source