message Documentation

Release 1.2.0

IIASA Energy Program

Aug 28, 2019

Contents
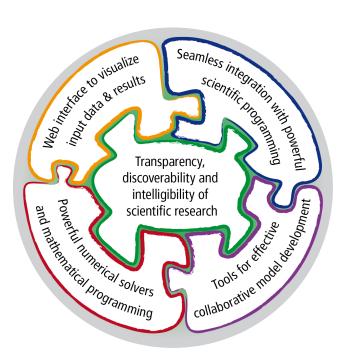
Fig. 1: The *ix modeling platform* (source: [1])

Overview and scope

MESSAGE*ix* is a versatile, open-source, dynamic systems-optimization modelling framework. It was developed for strategic energy planning and integrated assessment of energy-engineering-economy-environment systems (E4). The framework can be applied to analyse scenarios of the energy system transformation under technical-engineering constraints and political-societal considerations. The optimization model can be linked to the general-economy MACRO model to incorporate feedback between prices and demand levels for energy and commodities. The equations are implemented in the mathematical programming system GAMS for numerical solution of a model instance.

The MESSAGE*ix* framework is fully integrated with 's *ix modeling platform* (ixmp), a data warehouse for high-powered numerical scenario analysis. The platform supports an efficient workflow between original input data sources, the implementation of the mathematical model formulation, and the analysis of numerical results. The platform can be accessed via a web-based user interface and application programming interfaces (API) to the scientific programming languages Python and R. The platform also includes a generic data exchange API to GAMS for numerical computation.

This documentation provides an introduction and the mathematical formulation of the MESSAGE*ix* equations and auxiliary functions. For the scientific reference of the framework, see Huppmann et al. (2019) [1]. The formulation of MESSAGE*ix* is a re-implementation and extension of *'MESSAGE V'* (Messner and Strubegger, 1995 [5]), the Integrated Assessment model developed at the International Institute for Applied Systems Analysis (IIASA) since the 1980s. For an overview of the MESSAGE*ix* model used at the IIASA Energy Program and a list of recent publications, please refer to the MESSAGE-GLOBIOM documentation website.

Getting Started

2.1 Installation

2.1.1 Install GAMS

MESSAGE*ix* requires GAMS.

1. Download the latest version of GAMS for your operating system; run the installer.

2. Add GAMS to the `PATH` environment variable. This is **required** in order for MESSAGE*ix* to run the mathematical model core:

   - **on Windows, in the GAMS installer...**

     – Check the box labeled "Use advanced installation mode."

     – Check the box labeled "Add GAMS directory to PATH environment variable" on the Advanced Options page.

   - on macOS or Linux, add the following line to your `.bash_profile` (Mac) or `.bashrc` (Linux):

     ```
     export PATH=$PATH:/path/to/gams-directory-with-gams-binary
     ```

2.1.2 Install MESSAGE*ix* via Anaconda

After installing GAMS, we recommend that new users install Anaconda, and then use it to install MESSAGE*ix*. Advanced users may choose to install MESSAGE*ix* from source code (next section).

3. Install Python via Anaconda. We recommend the latest version, i.e., Python 3.6+.

4. Open a command prompt. We recommend Windows users use the "Anaconda Prompt" to avoid permissions issues when installing and using MESSAGE*ix*. This program is available in the Windows Start menu after installing Anaconda.

5. Install the `message-ix` package:

```
$ conda install -c conda-forge message-ix
```

2.1.3 Install MESSAGE*ix* from source

3. Install ixmp from source.

4. (Optional) If you intend to contribute changes to MESSAGE*ix*, first register a Github account, and fork the message_ix repository. This will create a new repository <user>/message_ix. (Please also see *Contributing to MESSAGEix development*.)

5. Clone either the main repository, or your fork; using the Github Desktop client, or the command line:

```
$ git clone git@github.com:iiasa/message_ix.git

# or:
$ git clone git@github.com:USER/message_ix.git
```

6. Open a command prompt in the message_ix directory and type:

```
$ pip install .
```

7. (Optional) Run the built-in test suite to check that MESSAGE*ix* functions correctly on your system:

```
$ pip install .[tests]
$ py.test tests
```

2.1.4 Configure model files

By default, the GAMS files containing the mathematical model core are installed with message_ix (e.g., in your Python site-packages directory). Many users will simply want to run MESSAGE*ix*, or use the Python or R APIs to manipulate data, parameters and scenarios. For these uses, direct editing of the GAMS files is not necessary.

To edit the files directly—to change the mathematical formulation, such as adding new types of parameters, constraints, etc.—use the messageix-config utility to place the model files in a directory of your choice:

```
$ messageix-config --model_path /path/to/model
```

Note: If you installed from source on Windows using install.bat, this command was run automatically, pointing to message_ix/model.

2.2 Tutorials

To get started with MESSAGE*ix*, the following tutorials are provided as Jupyter notebooks, which combine code, sample output, and explanatory text.

A static, non-interactive version of each notebook can be viewed online using the links below. In order to execute the tutorial code or make modifications, read the *Preparation* section, next.

2.2.1 Preparation

Getting tutorial files

If you installed MESSAGE*ix* from source, all notebooks are in the `tutorial` directory.

If you installed MESSAGE*ix* using Anaconda, download the notebooks using the `messageix-dl` utility. In a command prompt:

```
$ messageix-dl --local_path /path/to/tutorials
```

Running tutorials

Using Anaconda

The `nb_conda` package is required. It should be installed by default with Anaconda. If it was not, install it:

```
$ conda install nb_conda
```

1. Open "Jupyter Notebooks" from Anaconda's "Home" tab (or directly if you have the option).

2. Choose and open a tutorial notebook.

3. Each notebook requires a *kernel* that executes code interactively. Check that the kernel matches your conda environment, and if necessary change kernels with the menu, e.g. *Kernel → Change Kernel → Python [conda root]*.

From the command line

1. Navigate to the tutorial folder. For instance, if `messageix-dl` was used above:

   ```
   $ cd /path/to/tutorials
   ```

2. Start the Jupyter notebook:

   ```
   $ jupyter notebook
   ```

2.2.2 Westeros Electrified

This tutorial demonstrates how to model a very simple energy system, and then uses it to illustrate a range of framework features.

1. Build the baseline model.

2. Introduce emissions and a bound on the emissions.

3. Limit emissions using a tax instead of a bound.

4. Represent both coal and wind electricity, using a "firm capacity" formulation: each generation technology can supply some firm capacity, but the variable, renewable technology (wind) supplies less than coal.

5. Represent coal and wind electricity using a different, "flexibility requirement" formulation, wherein wind *requires* and coal *supplies* flexibility.

2.2.3 Austrian energy system

This tutorial demonstrates a stylized representation of a national electricity sector model, with several fossil and renewable power plant types.

1. Prepare the base model version, in Python or in R.

2. Plot results, in Python or in R.

3. Run a single policy scenario.

4. Run multiple policy scenarios. This tutorial has two notebooks: an introduction with some exercises and completed code for the exercises.

Have a question? First, check the *Frequently asked questions*, then try the community Google group:

- on the Web at https://groups.google.com/d/forum/message_ix, or

- via e-mail at <message_ix@googlegroups.com>.

Detailed documentation

3.1 MESSAGE*ix* framework overview

MESSAGE*ix* is a *framework* that can be used to develop and run many different *models*, each describing a different energy system. Models in the MESSAGE*ix* framework can range from very simple (as in the *Tutorials*) to highly detailed (e.g. the MESSAGE-GLOBIOM global model).

3.1.1 Supported features

The framework allows direct and explicit representation of:

- Energy **technologies** with arbitrary inputs and outputs, that can be used to describe a "reference energy system," including:

  - the fuel supply chain,

  - conversion technologies from primary to secondary energy forms,

  - transmission and distribution (e.g. of electricity), and

  - final demand for energy services.

- **Vintaging** of capacity, early retirement and decommissioning of technologies.

- System integration of **variable renewable energy sources** (based on Sullivan et al., 2013 [6] and Johnson et al., 2016 [2]).

- Soft relaxation of **dynamic constraints** on new capacity and activity (Keppo and Strubegger, 2010 [3]).

- **Perfect-foresight** and **dynamic-recursive** (myopic) solution algorithms.

3.1.2 Running a model
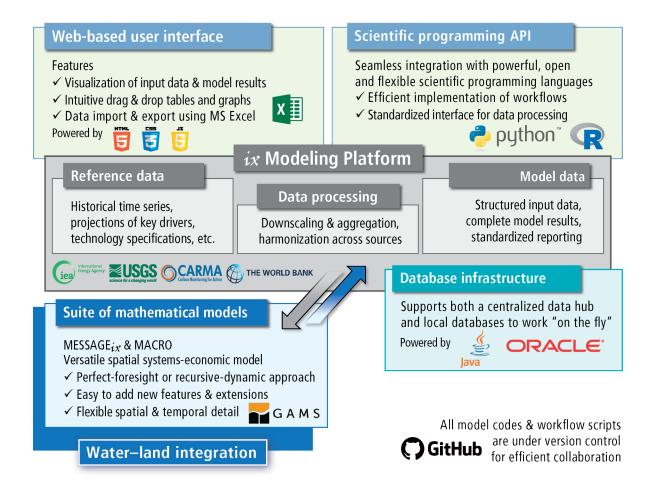
There are three ways to run a MESSAGE*ix* model:

Fig. 1: Components and their interlinkages in the *ix modeling platform* (source [1]): web-based user interface, scientific programming interface, modeling platform, database backend, implementation of the MESSAGE*ix* mathematical model formulation.

1. Via Python or R APIs using the packages/libraries *ixmp* and *message_ix*, calling *message_ix. Scenario.solve()*. (See the *Tutorials*.)

2. Using the file `MESSAGE_master.gms`, where the scenario name (i.e., the gdx input file), the optimization horizon (perfect foresight or myopic/ rolling-horizon version), and other options can be defined explicitly.

   This approach is recommended for users who prefer to work via GAMS IDE or other text editors to set the model specifications.

3. Directly from the command line calling the file `MESSAGE_run.gms` (see the auto-doc page). The scenario name and other arguments can be passed as command line parameters:

   ```
   $ gams MESSAGE_run.gms --in="<data-file>" --out="<output-file>"
   ```

Auto-generated documentation for the model run scripts is provided:

Note: This page is generated from inline documentation in `MESSAGE_run.gms`.

Run script for MESSAGE*ix* (stand-alone)

This is MESSAGE*ix* version 1.2.0. The version number must match the version number of the `ixmp` MESSAGE-scheme specifications used for exporting data and importing results.

This file contains the workflow of a MESSAGE*ix*-standalone run. It can be called:

- Via the scientific programming API's using the packages/libraries `ixmp` and `message_ix`, calling the method `solve()` of the `message_ix.Scenario` class (see the tutorials).

- using the file `MESSAGE_master.gms` with the option `$SETGLOBAL macromode "none"`, where the input data file name and other options are stated explicitly, or

- directly from the command line, with the input data file name and other options specific as command line parameters, e.g.

  ```
  gams MESSAGE_run.gms --in="<data-file>" [--out="<output-file>"]
  ```

By default, the data file (in gdx format) should be located in the `model/data` folder and be named in the format `MsgData_<name>.gdx`. Upon completion of the GAMS execution, a results file `<output-file>` will be written (or `model\output\MsgOutput.gdx` if `--out` is not provided).

Note: This page is generated from inline documentation in `MESSAGE-MACRO_run.gms`.

Run script for MESSAGE*ix* and MACRO

This is MESSAGE*ix*-MACRO version 1.2.0. The version number must match the version number of the `ixmp` `MESSAGE`-scheme specifications used for exporting data and importing results.

This file contains the workflow of a MESSAGE*ix*-MACRO run. It can be called:

- Via the scientific programming API's using the packages/libraries `ixmp` and `message_ix`, calling the method `solve()` of the `message_ix.Scenario` class (see the tutorials).

- using the file `MESSAGE_master.gms` with the option `$SETGLOBAL macromode "linked"`, where the input data file name and other options are stated explicitly, or

- directly from the command line, with the input data file name and other options specific as command line parameters, e.g.

```
gams MESSAGE-MACRO_run.gms --in="<data-file>" [--out="<output-file>"]
```

By default, the data file (in gdx format) should be located in the `model/data` folder and be named in the format `MsgData_<name>.gdx`. Upon completion of the GAMS execution, a results file `<output-file>` will be written (or `model\output\MsgOutput.gdx` if `--out` is not provided).

3.2 Python & R API

The application programming interface (API) for MESSAGE*ix* model developers is implemented in Python. Support for R usage of the core classes is provided through the reticulate package. For instance:

```
> library(reticulate)
> message <- library("message_ix")
> s <- messageix$Scenario(...)
```

3.2.1 `ixmp` package

`ixmp` provides three classes:

`Platform`([dbprops, dbtype, jvmargs])	Database-backed instance of the ixmp.
`TimeSeries`(mp, model, scenario[, version, . . .])	Generic collection of data in time series format.
`Scenario`(mp, model, scenario[, version, . . .])	Collection of model-related input and output data.

as well as some utility classes and methods:

`ixmp.config.Config`([read])	Configuration for ixmp.
`ixmp.model_settings.register_model`(name, config)	Register a new model with ixmp.
`ixmp.testing.make_dantzig`(mp[, solve])	Return `ixmp.Scenario` of Dantzig's canning/transport problem.

Core classes

class ixmp.**Platform**(*dbprops=None*, *dbtype=None*, *jvmargs=None*)
Database-backed instance of the ixmp.

Each Platform connects three components:

1. A **database** for storing model inputs and outputs. This may either be a local file (`dbtype='HSQLDB'`) or a database server accessed via a network connection. In the latter case, connection information is read from a *properties file*.

2. A Java Virtual Machine (**JVM**) to run core ixmp logic and access the database.

3. One or more **model(s)**, implemented in GAMS or another language or framework.

The constructor parameters control these components. `TimeSeries` and `Scenario` objects are specific to one Platform; to move data between platforms, see `Scenario.clone()`.

Parameters

- **dbprops** (`path-like, optional`) – If *dbtype* is `None`, the name of a database properties file (default: 'default.properties') in the properties file directory (default: ???) or the path of a properties file.

  If *dbtype* == *'HSQLDB'*, the path of a local database, (default: "$HOME/.local/ixmp/localdb/default") or name of a database file in the local database directory (default: "$HOME/.local/ixmp/localdb/").

- **dbtype** (`'HSQLDB', optional`) – Database type to use. If *None*, a remote database is accessed. If 'HSQLDB', a local database is created and used at the path given by *dbprops*.

- **jvmargs** (`str, optional`) – Options for launching the Java Virtual Machine, e.g., the maximum heap space: "-Xmx4G". See the JVM documentation for a list of options.

Scenario (*model*, *scen*, *version=None*, *scheme=None*, *annotation=None*, *cache=False*)

Initialize a new `Scenario`.

Deprecated since version 1.1.0: Instead, use:

```
>>> mp = ixmp.Platform(...)
>>> ixmp.Scenario(mp, ...)
```

add_region (*region*, *hierarchy*, *parent='World'*)

Define a region including a hierarchy level and a 'parent' region.

Tip: On a `Platform` backed by a shared database, a region may already exist with a different spelling. Use `regions()` first to check, and consider calling `add_region_synonym()` instead.

Parameters

- **region** (`str`) – Name of the region.

- **hierarchy** (`str`) – Hierarchy level of the region (e.g., country, R11, basin)

- **parent** (`str, default 'World'`) – Assign a 'parent' region.

add_region_synonym (*region*, *mapped_to*)

Define a synonym for a *region*.

When adding timeseries data using the synonym in the region column, it will be converted to *mapped_to*.

Parameters

- **region** (`str`) – Name of the region synonym.

- **mapped_to** (`str`) – Name of the region to which the synonym should be mapped.

add_unit (*unit*, *comment='None'*)

Define a unit.

Parameters

- **unit** (`str`) – Name of the unit.

- **comment** (`str, optional`) – Annotation describing the unit or why it was added. The current database user and timestamp are appended automatically.

check_access (*user*, *models*, *access='view'*)

Check access to specific model

Parameters

- **user** (`str`) – Registered user name

- **models** (`str or list of str`) – Model(s) name

- **access** (`str, optional`) – Access type - view or edit

close_db()
:   Close the database connection.

    A HSQL database can only be used by one `Platform` instance at a time. Any existing connection must be closed before a new one can be opened.

open_db()
:   (Re-)open the database connection.

    The database connection is opened automatically for many operations. After calling `close_db()`, it must be re-opened.

regions()
:   Return all regions defined for the IAMC-style timeseries format including known synonyms.

    Returns

    Return type pandas.DataFrame

scenario_list(*default=True*, *model=None*, *scen=None*)
:   Return information on TimeSeries and Scenarios in the database.

    Parameters

    - **default** (`bool, optional`) – Return *only* the default version of each Time-Series/Scenario (see `TimeSeries.set_as_default()`). Any (*model*, *scenario*) without a default version is omitted. If `False`, return all versions.

    - **model** (`str, optional`) – A model name. If given, only return information for *model*.

    - **scen** (`str, optional`) – A scenario name. If given, only return information for *scen*.

    Returns

    Scenario information, with the columns:

    - model, scenario, version, and scheme—Scenario identifiers; see `Scenario`.

    - is_default—`True` if the version is the default version for the (model, scenario).

    - is_locked—`True` if the Scenario has been locked for use.

    - cre_user and cre_date—database user that created the Scenario, and creation time.

    - upd_user and upd_date—user and time for last modification of the Scenario.

    - lock_user and lock_date—user that locked the Scenario and lock time.

    - annotation: description of the Scenario or changelog.

    Return type pandas.DataFrame

set_log_level(*level*)
:   Set global logger level (for both Python and Java)

    Parameters **level** (`str`) – set the logger level if specified, see https://docs.python.org/3/library/logging.html#logging-levels

units()
> Return all units described in the database.

>> **Returns**

>> **Return type** list

class ixmp.**TimeSeries**(*mp*, *model*, *scenario*, *version=None*, *annotation=None*)
> Generic collection of data in time series format.

> TimeSeries is the parent/super-class of *Scenario*.

> A TimeSeries is uniquely identified on its *Platform* by three values:

> 1. *model*: the name of a model used to perform calculations between input and output data.

>> • In TimeSeries storing non-model data, arbitrary strings can be used.

>> • In a *Scenario*, the *model* is a reference to a GAMS program registered to the *Platform* that can be solved with *Scenario.solve()*. See *ixmp.model_settings.register_model()*.

> 2. *scenario*: the name of a specific, coherent description of the real- world system being modeled. Any *model* may be used to represent mutiple alternate, or 'counter-factual', *scenarios*.

> 3. *version*: an integer identifying a specific iteration of a (*model*, *scenario*). A new *version* is created by:

>> • Instantiating a new TimeSeries with the same *model* and *scenario* as an existing TimeSeries.

>> • Calling *Scenario.clone()*.

> Optionally, one *version* may be set as a **default version**. See *set_as_default()*.

>> **Parameters**

>>> • **mp** (*Platform*) – ixmp instance in which to store data.

>>> • **model** (*str*) – Model name.

>>> • **scenario** (*str*) – Scenario name.

>>> • **version** (*int or str, optional*) – If omitted and a default version of the (*model*, *scenario*) has been designated (see *set_as_default()*), load that version. If *int*, load a specific version. If `'new'`, create a new TimeSeries.

>>> • **annotation** (*str, optional*) – A short annotation/comment used when version='new'.

add_timeseries(*df*, *meta=False*)
> Add data to the TimeSeries.

>> **Parameters**

>>> • **df** (`pandas.DataFrame`) – Data to add. *df* must have the following columns:

>>>> – *region* or *node*

>>>> – *variable*

>>>> – *unit*

>>> Additional column names may be either of:

>>>> – *year* and *value*—long, or 'tabular', format.

>>>> – one or more specific years—wide, or 'IAMC' format.

- **meta** (*bool, optional*) – If `True`, store *df* as metadata. Metadata is treated specially when `Scenario.clone()` is called for Scenarios created with `scheme='MESSAGE'`.

check_out(*timeseries_only=False*)
> check out from the ixmp database instance for making changes

commit(*comment*)
> Commit all changed data to the database.
>
> `version` is not incremented.

discard_changes()
> Discard all changes and reload from the database.

is_default()
> Return `True` if the `version` is the default version.

last_update()
> get the timestamp of the last update/edit of this TimeSeries

preload_timeseries()
> Preload timeseries data to in-memory cache. Useful for bulk updates.

remove_timeseries(*df*)
> Remove timeseries data from the TimeSeries instance.
>
> > **Parameters df** (`pandas.DataFrame`) – Data to remove. *df* must have the following columns:
> >
> > - *region* or *node*
> > - *variable*
> > - *unit*
> > - *year*

run_id()
> get the run id of this TimeSeries

set_as_default()
> Set the current `version` as the default.

timeseries(*iamc=False*, *region=None*, *variable=None*, *level=None*, *unit=None*, *year=None*, ***kwargs*)
> Retrieve TimeSeries data.
>
> > **Parameters**
> >
> > - **iamc** (`bool, default:  False`) – Return data in wide/'IAMC' format. If `False`, return data in long/'tabular' format; see `add_timeseries()`.
> > - **region** (`str or list of strings`) – Regions to include in returned data.
> > - **variable** (`str or list of strings`) – Variables to include in returned data.
> > - **unit** (`str or list of strings`) – Units to include in returned data.
> > - **year** (`str, int or list of strings or integers`) – Years to include in returned data.
> >
> > **Returns** Specified data.
> >
> > **Return type** `pandas.DataFrame`

version()
>   get the version number of this TimeSeries

class ixmp.**Scenario**(*mp*, *model*, *scenario*, *version=None*, *scheme=None*, *annotation=None*, *cache=False*)
>   Collection of model-related input and output data.
>
>   A Scenario is a *TimeSeries* associated with a particular model that can be run on the current *Platform* by calling *solve()*. The Scenario also stores the output, or 'solution' of a model run; this includes the 'level' and 'marginal' values of GAMS equations and variables.
>
>   Data in a Scenario are closely related to different types in the GAMS data model:
>
>   - A **set** is a named collection of labels. See *init_set()*, *add_set()*, and *set()*. There are two types of sets:
>
>   1. Sets that are lists of labels.
>
>   2. Sets that are 'indexed' by one or more other set(s). For this type of set, each member is an ordered tuple of the labels in the index sets.
>
>   - A **scalar** is a named, single, numerical value. See *init_scalar()*, *change_scalar()*, and *scalar()*.
>
>   - **Parameters**, **variables**, and **equations** are multi-dimensional arrays of values that are indexed by one or more sets (i.e. with dimension 1 or greater). The Scenario methods for handling these types are very similar; they mainly differ in how they are used within GAMS models registered with ixmp:
>
>   - **Parameters** are generic data that can be defined before a model run. They may be altered by the model solution. See *init_par()*, *remove_par()*, *par_list()*, *add_par()*, and *par()*.
>
>   - **Variables** are calculated during or after a model run by GAMS code, so they cannot be modified by a Scenario. See *init_var()*, *var_list()*, and *var()*.
>
>   - **Equations** describe fundamental relationships between other types (parameters, variables, and scalars) in a model. They are defined in GAMS code, so cannot be modified by a Scenario. See *init_equ()*, *equ_list()*, and *equ()*.
>
>   Parameters
>
>   - **mp** (*Platform*) – ixmp instance in which to store data.
>
>   - **model** (*str*) – Model name; must be a registered model.
>
>   - **scenario** (*str*) – Scenario name.
>
>   - **version** (*str or int or at.ac.iiasa.ixmp.objects.Scenario, optional*) – If omitted, load the default version of the (*model*, *scenario*). If *int*, load the specified version. If 'new', initialize a new Scenario.
>
>   - **scheme** (*str, optional*) – Use an explicit scheme for initializing a new scenario.
>
>   - **annotation** (*str, optional*) – A short annotation/comment used when version='new'.
>
>   - **cache** (*bool, optional*) – Store data in memory and return cached values instead of repeatedly querying the database.

add_par(*name*, *key*, *val=None*, *unit=None*, *comment=None*)
>   Set the values of a parameter.
>
>   Parameters
>
>   - **name** (*str*) – Name of the parameter.

- **key** (*str, list/range of strings/values, dictionary, dataframe*) – element(s) to be added

- **val** (*values, list/range of values, optional*) – element values (only used if 'key' is a string or list/range)

- **unit** (*str, list/range of strings, optional*) – element units (only used if 'key' is a string or list/range)

- **comment** (*str or list/range of strings, optional*) – comment (optional, only used if 'key' is a string or list/range)

add_set (*name*, *key*, *comment=None*)
    Add elements to an existing set.

    Parameters

- **name** (*str*) – Name of the set.

- **key** (str or iterable of str or dict or `pandas.DataFrame`) – Element(s) to be added. If *name* exists, the elements are appended to existing elements.

- **comment** (*str or iterable of str, optional*) – Comment describing the element(s). Only used if *key* is a string or list/range.

    Raises `jpype.JavaException` – If the set *name* does not exist. `init_set()` must be called before `add_set()`.

change_scalar (*name*, *val*, *unit*, *comment=None*)
    Set the value and unit of a scalar.

    Parameters

- **name** (*str*) – Name of the scalar.

- **val** (*number*) – New value of the scalar.

- **unit** (*str*) – New unit of the scalar.

- **comment** (*str, optional*) – Description of the change.

clear_cache (*name=None*, *ix_type=None*)
    clear the Python cache of item elements

    Parameters

- **name** (*str, optional*) – item name (*None* clears entire Python cache)

- **ix_type** (*str, optional*) – type of item (if provided, cache clearing is faster)

clone (*model=None*, *scenario=None*, *annotation=None*, *keep_solution=True*, *first_model_year=None*, *platform=None*, ***kwargs*)
Clone the current scenario and return the clone.

If the (*model*, *scenario*) given already exist on the *Platform*, the *version* for the cloned Scenario follows the last existing version. Otherwise, the *version* for the cloned Scenario is 1.

Note: *clone()* does not set or alter default versions. This means that a clone to new (*model*, *scenario*) names has no default version, and will not be returned by *Platform.scenario_list()* unless *default=False* is given.

    Parameters

- **model** (*str, optional*) – New model name. If not given, use the existing model name.

- **scenario** (*str, optional*) – New scenario name. If not given, use the existing scenario name.

- **annotation** (*str, optional*) – Explanatory comment for the clone commit message to the database.

- **keep_solution** (*bool, optional*) – If True, include all timeseries data and the solution (vars and equs) from the source scenario in the clone. If False, only include timeseries data marked *meta=True* (see *TimeSeries.add_timeseries()*).

- **first_model_year** (*int, optional*) – If given, all timeseries data in the Scenario is omitted from the clone for years from *first_model_year* onwards. Timeseries data with the *meta* flag (see *TimeSeries.add_timeseries()*) are cloned for all years.

- **platform** (*Platform*, optional) – Platform to clone to (default: current platform)

equ (*name*, *filters=None*, ***kwargs*)
> return a dataframe of (filtered) elements for a specific equation

> **Parameters**

> - **name** (*str*) – name of the equation

> - **filters** (*dict*) – index names mapped list of index set elements

equ_list ()
> List all defined equations.

get_meta (*name=None*)
> get scenario metadata

> **Parameters name** (*str, optional*) – metadata attribute name

has_equ (*name*)
> check whether the scenario has an equation with that name

has_par (*name*)
> check whether the scenario has a parameter with that name

has_set (*name*)
> check whether the scenario has a set with that name

has_solution ()
> Return True if the Scenario has been solved.

> If has_solution() == True, model solution data exists in the db.

has_var (*name*)
> check whether the scenario has a variable with that name

idx_names (*name*)
> return the list of index names for an item (set, par, var, equ)

> **Parameters name** (*str*) – name of the item

idx_sets (*name*)
> Return the list of index sets for an item (set, par, var, equ)

> **Parameters name** (*str*) – name of the item

init_equ (*name*, *idx_sets=None*, *idx_names=None*)
> Initialize a new equation.

> **Parameters**
>
> - **name** (`str`) – name of the item
> - **idx_sets** (`list of str`) – index set list
> - **idx_names** (`list of str, optional`) – index name list

init_par(*name*, *idx_sets*, *idx_names=None*)

> Initialize a new parameter.
>
> **Parameters**
>
> - **name** (`str`) – Name of the parameter.
> - **idx_sets** (`list of str`) – Names of sets that index this parameter.
> - **idx_names** (`list of str, optional`) – Names of the dimensions indexed by *idx_sets*.

init_scalar(*name*, *val*, *unit*, *comment=None*)

> Initialize a new scalar.
>
> **Parameters**
>
> - **name** (`str`) – Name of the scalar
> - **val** (`number`) – Initial value of the scalar.
> - **unit** (`str`) – Unit of the scalar.
> - **comment** (`str, optional`) – Description of the scalar.

init_set(*name*, *idx_sets=None*, *idx_names=None*)

> Initialize a new set.
>
> **Parameters**
>
> - **name** (`str`) – Name of the set.
> - **idx_sets** (`list of str, optional`) – Names of other sets that index this set.
> - **idx_names** (`list of str, optional`) – Names of the dimensions indexed by *idx_sets*.
>
> **Raises** `jpype.JavaException` – If the set (or another object with the same *name*) already exists.

init_var(*name*, *idx_sets=None*, *idx_names=None*)

> initialize a new variable in the scenario
>
> **Parameters**
>
> - **name** (`str`) – name of the item
> - **idx_sets** (`list of str`) – index set list
> - **idx_names** (`list of str, optional`) – index name list

load_scenario_data()

> Load all Scenario data into memory.
>
> **Raises** `ValueError` – If the Scenario was instantiated with `cache=False`.

par(*name*, *filters=None*, ***kwargs*)

> return a dataframe of (filtered) elements for a specific parameter
>
> **Parameters**

- **name** (*str*) – name of the parameter

- **filters** (*dict*) – index names mapped list of index set elements

par_list()
:   List all defined parameters.

read_sol_from_gdx(*path*, *filename*, *comment=None*, *var_list=None*, *equ_list=None*, *check_solution=True*)
:   read solution from GAMS gdx and import it to the scenario

    Parameters

    - **path** (*str*) – path to the folder

    - **filename** (*str*) – name of the gdx file

    - **comment** (*str*) – comment to be added to the changelog

    - **var_list** (*list of str*) – variables (levels and marginals) to be imported from gdx

    - **equ_list** (*list of str*) – equations (levels and marginals) to be imported from gdx

    - **check_solution** (*boolean, optional*) – raise an error if GAMS did not solve to optimality (only applicable for a MESSAGE-scheme scenario)

remove_par(*name*, *key=None*)
:   Remove parameter values or an entire parameter.

    Parameters

    - **name** (*str*) – Name of the parameter.

    - **key** (*dataframe or key list or concatenated string, optional*) – elements to be removed

remove_set(*name*, *key=None*)
:   delete a set from the scenario or remove an element from a set (if key is specified)

    Parameters

    - **name** (*str*) – name of the set

    - **key** (*dataframe or key list or concatenated string*) – elements to be removed

remove_solution(*first_model_year=None*)
:   Remove the solution from the scenario

    This function removes the solution (variables and equations) and timeseries data marked as *meta=False* from the scenario (see *TimeSeries.add_timeseries()*).

    Parameters **first_model_year** (*int, optional*) – If given, timeseries data marked as *meta=False* is removed only for years from *first_model_year* onwards.

    Raises *ValueError* – If Scenario has no solution or if *first_model_year* is not *int*.

scalar(*name*)
:   Return the value and unit of a scalar.

    Parameters **name** (*str*) – Name of the scalar.

    Returns {'value'

    Return type value, 'unit': unit}

set(*name*, *filters=None*, ***kwargs*)
:   Return the (filtered) elements of a set.

> **Parameters**
>
> - **name** (`str`) – Name of the set.
> - **filters** (`dict`) – Mapping of *dimension_name* → *elements*, where *dimension_name* is one of the *idx_names* given when the set was initialized (see `init_set()`), and *elements* is an iterable of labels to include in the return value.
>
> **Returns**
>
> **Return type** pandas.DataFrame

set_list()
> List all defined sets.

set_meta(*name*, *value*)
> set scenario metadata
>
> > **Parameters**
> >
> > - **name** (`str`) – metadata attribute name
> > - **value** (`str or number or bool`) – metadata attribute value

solve(*model, case=None, model_file=None, in_file=None, out_file=None, solve_args=None, comment=None, var_list=None, equ_list=None, check_solution=True, callback=None, gams_args=['LogOption=4'], cb_kwargs={}*)
> Solve the model and store output.
>
> ixmp 'solves' a model using the following steps:
>
> 1. Write all Scenario data to a GDX model input file.
> 2. Run GAMS for the specified *model* to perform calculations.
> 3. Read the model output, or 'solution', into the database.
>
> If the optional argument *callback* is given, then additional steps are performed:
>
> 4. Execute the *callback* with the Scenario as an argument. The Scenario has an *iteration* attribute that stores the number of times the underlying model has been solved (#2).
> 5. If the *callback* returns `False` or similar, go to #1; otherwise exit.
>
> > **Parameters**
> >
> > - **model** (`str`) – model (e.g., MESSAGE) or GAMS file name (excluding '.gms')
> > - **case** (`str`) – identifier of gdx file names, defaults to 'model_name_scen_name'
> > - **model_file** (`str, optional`) – path to GAMS file (including '.gms' extension)
> > - **in_file** (`str, optional`) – path to GAMS gdx input file (including '.gdx' extension)
> > - **out_file** (`str, optional`) – path to GAMS gdx output file (including '.gdx' extension)
> > - **solve_args** (`str, optional`) – arguments to be passed to GAMS (input/output file names, etc.)
> > - **comment** (`str, optional`) – additional comment added to changelog when importing the solution
> > - **var_list** (`list of str, optional`) – variables to be imported from the solution

- **equ_list** (*list of str, optional*) – equations to be imported from the solution

- **check_solution** (*boolean, optional*) – flag whether a non-optimal solution raises an exception (only applies to MESSAGE runs)

- **callback** (*callable, optional*) – Method to execute arbitrary non-model code. Must accept a single argument, the Scenario. Must return a non-`False` value to indicate convergence.

- **gams_args** (*list of str, optional*) – Additional arguments for the CLI call to GAMS. See, e.g., https://www.gams.com/latest/docs/UG_GamsCall.html#UG_GamsCall_ListOfCommandLineParameters

  - *LogOption=4* prints output to stdout (not console) and the log file.

- **cb_kwargs** (*dict, optional*) – Keyword arguments to pass to *callback*.

Warns UserWarning – If *callback* is given and returns `None`. This may indicate that the user has forgotten a `return` statement, in which case the iteration will continue indefinitely.

Raises `ValueError` – If the Scenario has already been solved.

to_gdx(*path*, *filename*, *include_var_equ=False*)
    export the scenario data to GAMS gdx

    Parameters

- **path** (*str*) – path to the folder

- **filename** (*str*) – name of the gdx file

- **include_var_equ** (*boolean, optional*) – indicator whether to include variables/equations in gdx

var(*name*, *filters=None*, ***kwargs*)
    return a dataframe of (filtered) elements for a specific variable

    Parameters

- **name** (*str*) – name of the variable

- **filters** (*dict*) – index names mapped list of index set elements

var_list()
    List all defined variables.

years_active(*node*, *tec*, *yr_vtg*)
    return a list of years in which a technology of certain vintage at a specific node can be active

    Parameters

- **node** (*str*) – node name

- **tec** (*str*) – name of the technology

- **yr_vtg** (*str*) – vintage year

Utility methods

class ixmp.config.**Config**(*read=True*)
    Configuration for ixmp.

    When imported, *ixmp* reads a configuration file *config.json* in the first of the following directories:

1. *IXMP_DATA*, if defined.

2. *${XDG_DATA_HOME}/ixmp*, if defined.

3. *$HOME/.local/share/ixmp*.

4. *$HOME/.local/ixmp* (used by ixmp <= 1.1).

The file may define either or both of the following configuration keys, in JSON format:

- *DB_CONFIG_PATH*: location for database properties files. A `ixmp.Platform` instantiated with a relative path name for the *dbprops* argument will locate the file first in the current working directory, then in *DB_CONFIG_PATH*, then in the four directories above.

- *DEFAULT_DBPROPS_FILE*: path to a default database properties file. A `ixmp.Platform` instantiated with no arguments will use this file.

- *DEFAULT_LOCAL_DB_PATH*: path to a directory where a local directory should be created. A `ixmp.Platform` instantiated with *dbtype='HSQLDB'* will create or reuse a database in this path.

> **Parameters read** (`bool`) – Read *config.json* on startup.

clear()
> Clear all configuration keys by setting their values to None.

find_dbprops(*fname*)
> Return the absolute path to a database properties file.
>
> Searches for a file named *fname*, first in the current working directory (.), then in the ixmp default location.
>
> > **Parameters fname** (`str`) – Name of a database properties file to locate.
> >
> > **Returns** Absolute path to *fname*.
> >
> > **Return type** str
> >
> > **Raises** `FileNotFoundError` – *fname* is not found in any of the search paths.

get(*key*)
> Return the value of a configuration *key*.

read()
> Try to read configuration keys from file.
>
> If successful, the configuration key 'CONFIG_PATH' is set to the path of the file.

save()
> Write configuration keys to file.
>
> *config.json* is created in the first of the ixmp configuration directories that exists. Only non-null values are written.

set(*key*, *value*)
> Set configuration *key* to *value*.

ixmp.model_settings.**register_model**(*name*, *config*)
> Register a new model with ixmp.
>
> > **Parameters**
> >
> > - **name** (`str`) – Model name.
> >
> > - **config** (`dict`) – Configuration settings for the model, with the following keys. Each key is a format string that may use certain named values:

- *model_file* (`str`): GAMS source file (`.gms`) containing the model. E.g. "{model}_run.gms". Available values: model, case.

- *inp* (`str`): input path; location where the model expects a GDX file containing input data. E.g. "{model}_{case}_in.gdx". Available values: model, case.

- *outp* (`str`): output path; location where the model will create a GDX file containing output data. E.g. "{model}_{case}_out.gdx". Available values: model, case.

- *args* (`list` of `str`): additional GAMS command-line args to be passed when invoking the model. Available values: model, case, inp, outp.

The *model* and *case* formatting values are generated from `ixmp.Scenario.model` and `ixmp.Scenario.scenario`, respectively, with spaces (" ") converted to underscores ("_").

Utilities for testing ixmp.

These include:

- pytest hooks, and 3 fixtures: tmp_env, test_mp, and test_mp_props.

- Methods for setting up and populating test ixmp databases: create_local_testdb() and dantzig_transport().

- Methods to run and retrieve values from Jupyter notebooks: run_notebook() and get_cell_output().

`ixmp.testing.`**`make_dantzig`**(*mp*, *solve=False*)
    Return *ixmp.Scenario* of Dantzig's canning/transport problem.

   > **Parameters**
   >
   >   - **mp** (`ixmp.Platform`) – Platform on which to create the scenario.
   >
   >   - **solve** (*bool or os.PathLike*) – If not `False`, then *solve* is interpreted as a path to a directory, and the model `transport_ixmp.gms` in the directory is run for the scenario.

3.2.2 `message_ix` package

MESSAGE*ix* models are created using the *message_ix.Scenario* class. Several utility methods are also provided in the module *message_ix.utils*.

`core.`**`DEFAULT_SOLVE_OPTIONS = {'advind': 0, 'epopt': 1e-06, 'lpmethod': 2, 'threads': 4}`**
    Solver options used by *message_ix.Scenario.solve()*. These configure the GAMS CPLEX solver (or another solver, if selected); see the solver documentation for possible values.

class `message_ix.`**`Scenario`**(*mp*, *model*, *scenario=None*, *version=None*, *annotation=None*, *cache=False*)
    MESSAGE*ix* Scenario.

   This class extends *ixmp.Scenario* and inherits all its methods. It defines additional methods specific to MESSAGE*ix*.

   `add_cat`(*name*, *cat*, *keys*, *is_unique=False*)
       Map elements from *keys* to category *cat* within set *name*.

      > **Parameters**
      >
      >   - **name** (*str*) – Name of the set.
      >
      >   - **cat** (*str*) – Name of the category.
      >
      >   - **keys** (*str or list of str*) – Element keys to be added to the category mapping.

- **is_unique** (*bool, optional*) – If *True*, then *cat* must have only one element. An exception is raised if *cat* already has an element, or if `len(keys) > 1`.

add_horizon(*data*)
  Add sets related to temporal dimensions of the model.

> Parameters **data** (*dict-like*) – Year sets. "year" is a required key. "firstmodelyear" is optional; if not provided, the first element of "year" is used.

Examples

```
>>> s = message_ix.Scenario()
>>> s.add_horizon({'year': [2010, 2020]})
>>> s.add_horizon({'year': [2010, 2020], 'firstmodelyear': 2020})
```

add_spatial_sets(*data*)
  Add sets related to spatial dimensions of the model.

> Parameters **data** (*dict*) – Mapping of *level* → *member*. Each member may be:
>
> - A single label for elements.
>
> - An iterable of labels for elements.
>
> - A recursive `dict` following the same convention, defining sub-levels and their members.

Examples

```
>>> s = message_ix.Scenario()
>>> s.add_spatial_sets({'country': 'Austria'})
>>> s.add_spatial_sets({'country': ['Austria', 'Germany']})
>>> s.add_spatial_sets({'country': {
...     'Austria': {'state': ['Vienna', 'Lower Austria']}}})
```

cat(*name*, *cat*)
  return a list of all set elements mapped to a category

> **Parameters**
>
> - **name** (*string*) – name of the set
>
> - **cat** (*string*) – name of the category

cat_list(*name*)
  return a list of all categories for a set

> Parameters **name** (*string*) – name of the set

clone(*model=None*, *scenario=None*, *annotation=None*, *keep_solution=True*, *shift_first_model_year=None*, *platform=None*)
  Clone the current scenario and return the clone.

  If the (*model*, *scenario*) given already exist on the `Platform`, the *version* for the cloned Scenario follows the last existing version. Otherwise, the *version* for the cloned Scenario is 1.

  Note: *clone()* does not set or alter default versions. This means that a clone to new (*model*, *scenario*) names has no default version, and will not be returned by `Platform.scenario_list()` unless *default=False* is given.

Parameters

- **model** (`str, optional`) – New model name. If not given, use the existing model name.

- **scenario** (`str, optional`) – New scenario name. If not given, use the existing scenario name.

- **annotation** (`str, optional`) – Explanatory comment for the clone commit message to the database.

- **keep_solution** (`bool, default True`) – If `True`, include all timeseries data and the solution (vars and equs) from the source scenario in the clone. Otherwise, only timeseries data marked as *meta=True* (see `TimeSeries.add_timeseries()`) or prior to *first_model_year* (see `TimeSeries.add_timeseries()`) are cloned.

- **shift_first_model_year** (`int, optional`) – If given, the values of the solution are transfered to parameters *historical_*, parameter *resource_volume* is updated, and the *first_model_year* is shifted. The solution is then discarded, see `TimeSeries.remove_solution()`.

- **platform** (`Platform`, optional) – Platform to clone to (default: current platform)

firstmodelyear
    Returns the first model year of the scenario.

read_excel(*fname*, *add_units=False*, *commit_steps=False*)
    Read Excel file data and load into the scenario.

    Parameters

    - **fname** (`string`) – path to file

    - **add_units** (`bool`) – add missing units, if any, to the platform instance. default: False

    - **commit_steps** (`bool`) – commit changes after every data addition. default: False

remove_solution()
    Remove the solution from the scenario

    Delete the model solution (variables and equations) and timeseries data marked as *meta=False* (see `TimeSeries.add_timeseries()`) prior to the first model year.

rename(*name*, *mapping*, *keep=False*)
    Rename an element in a set

    Parameters

    - **name** (`str`) – name of the set to change (e.g., 'technology')

    - **mapping** (`str`) – mapping of old (current) to new set element names

    - **keep** (`bool, optional, default: False`) – keep the old values in the model

solve(*model='MESSAGE'*, *solve_options={}*, ***kwargs*)
    Solve the model for the Scenario.

    By default, *ixmp.Scenario.solve()* is called with 'MESSAGE' as the *model* argument; see the documentation of that method for other arguments. *model* may also be overwritten, e.g.:

    ```
    >>> s.solve(model='MESSAGE-MACRO')
    ```

    Parameters

- **model** (`str, optional`) – Type of model to solve, e.g. 'MESSAGE' or 'MESSAGE-MACRO'.

- **solve_options** (`dict, optional`) – name, value pairs to use for GAMS solver optfile. See `message_ix.DEFAULT_SOLVE_OPTIONS` for defaults and https://www.gams.com/latest/docs/S_CPLEX.html for possible keys.

to_excel (*fname*)

Save a scenario as an Excel file. NOTE: Cannot export solution currently (only model data) due to limitations in excel sheet names (cannot have multiple sheet names which are identical except for upper/lower case).

> **Parameters** **fname** (`string`) – path to file

vintage_and_active_years (*ya_args=None*, *in_horizon=True*)

Return sets of vintage and active years for use in data input.

For a valid pair *(year_vtg, year_act)*, the following conditions are satisfied:

1. Both the vintage year (*year_vtg*) and active year (*year_act*) are in the model's `year` set.

2. *year_vtg <= year_act*.

3. *year_act <=* the model's first year **or** *year_act* is in the smaller subset *ixmp.Scenario. years_active()* for the given *ya_args*.

> **Parameters**
>
> - **ya_args** (`tuple of (node, technology, year_vtg), optional`) – Arguments to *ixmp.Scenario.years_active()*.
>
> - **in_horizon** (`bool, optional`) – Restrict years returned to be within the current model horizon.
>
> **Returns** with columns, "year_vtg" and "year_act", in which each row is a valid pair.
>
> **Return type** pandas.DataFrame

Utility methods

message_ix.utils.**make_df** (*base*, ***kwargs*)

Extend or overwrite *base* with new values from *kwargs*.

> **Parameters**
>
> - **base** (dict, `pandas.Series`, or `pandas.DataFrame`) – Existing dataset to append to.
>
> - ****kwargs** – Additional values to append to *base*.
>
> **Returns** *base* modified with *kwargs*.
>
> **Return type** `pandas.DataFrame`

Examples

Scalar values in *base* or *kwargs* are broadcast. The number of rows in the returned `pandas.DataFrame` equals the length of the longest item in either argument.

```
>>> base = {'foo': 'bar'}
>>> make_df(base, baz=[42, 43, 44])
    foo    baz
0   bar     42
1   bar     43
2   bar     44
```

message_ix.utils.**make_ts**(*df*, *time_col*, *value_col*, *metadata={}*)
> The function groups the dataframe by the year specified in year_col_name (year_act Vs. year_vtg). It then reshapes the dataframe df to reseble the timeseries requirements: sets the unit, the variable name, and the value column to the one specified in value_col_name. it further drops all all additional columns.

message_ix.utils.**matching_rows**(*df*, *row*, *match_columns=[]*)
> The function finds all the columns in a dataframe that are specified in the match columns list.

message_ix.utils.**multiply_df**(*df1*, *column1*, *df2*, *column2*)
> The function merges dataframe df1 with df2 and multiplies column1 with column2. The function returns the new merged dataframe with the result of the muliplication in the column 'product'.

message_ix.testing.**make_dantzig**(*mp*, *solve=False*, *multi_year=False*)
> Return an *message_ix.Scenario* for Dantzig's canning problem.

> > **Parameters**
> >
> > - **mp** (`ixmp.Platform`) – Platform on which to create the scenario.
> >
> > - **solve** (*bool, optional*) – If True, the scenario is solved.
> >
> > - **multi_year** (*bool, optional*) – If True, the scenario has years 1963–1965 inclusive. Otherwise, the scenario has the single year 1963.

message_ix.testing.**make_westeros**(*mp*, *emissions=False*, *solve=False*)
> Return an *message_ix.Scenario* for the Westeros model.

> This is the same model used in the `westeros_baseline.ipynb` tutorial.

> > **Parameters**
> >
> > - **mp** (`ixmp.Platform`) – Platform on which to create the scenario.
> >
> > - **emissions** (*bool, optional*) – If True, the `emissions_factor` parameter is also populated for CO2.
> >
> > - **solve** (*bool, optional*) – If True, the scenario is solved.

3.3 Mathematical specification

These pages provide comprehensive description of the variables and equations in the core MESSAGEix mathematical implementation.

Note: This page is generated from inline documentation in `MESSAGE/sets_maps_def.gms`.

3.3.1 Sets and mappings definition

This file contains the definition of all sets and mappings used in MESSAGE*ix*.

Sets in the MESSAGE*ix* implementation

Set name	Notation	Explanatory comments
node[1]	$n \in N$	regions, countries, grid cells
commodity	$c \in C$	resources, electricity, water, land availability, etc.
level	$l \in L$	levels of the reference energy system or supply chain (primary, secondary, … , useful)
grade	$g \in G$	grades of resource quality in the extraction & mining sector
technology [tec]	$t \in T$	technologies that use input commodities to produce outputs; the short-hand notation "tec" is used in the GAMS implementation
mode[2]	$m \in M$	modes of operation for specific technologies
emission	$e \in E$	greenhouse gases, pollutants, etc.
land_scenario	$s \in S$	scenarios of land use (for land-use model emulator)
land_type	$u \in U$	land-use types (e.g., field, forest, pasture)
year [year_all][34]	$y \in Y$	model horizon (including historical periods for vintage structure of installed capacity and dynamic constraints)
time[5]	$h \in H$	subannual time periods (seasons, days, hours)
relation[6]	$r \in R$	set of generic linear constraints
rating	$q \in Q$	identifies the 'quality' of the renewable energy potential
lvl_spatial		set of spatial hierarchy levels (global, region, country, grid cell)
lvl_temporal		set of temporal hierarchy levels (year, season, day, hour)

Category types and mappings

This feature is used to easily implement aggregation across groups of set elements. For example, by setting an upper bound over an emission type, the constraint enforces that the sum over all emission species mapped to that type via the mapping set `cat_emission` satisfies that upper bound.

---

[1] The set `node` includes spatial units across all levels of spatial disaggregation (global, regions, countries, basins, grid cells). The hierarchical mapping is implemented via the mapping set `map_spatial_hierarchy`. This set always includes an element 'World' when initializing a `MESSAGE`-scheme `message_ix.Scenario`.

[2] For example, high electricity or high heat production modes of operation for combined heat and power plants.

[3] In the MESSAGE*ix* implementation in GAMS, the set `year_all` denotes the "superset" of the entire horizon (historical and model horizon), and the set `year` is a dynamic subset of `year_all`. This facilitates an efficient implementation of the historical capacity build-up and the (optional) recursive-dynamic solution approach. When working with a `message_ix.Scenario` via the scientific programming API, the set of all periods is called `year` for a more concise notation. The specification of the model horizon is implemented using the mapping set `cat_year` and the type "firstmodelyear".

[4] In MESSAGE*ix*, the key of an element in set `year` identifies *the last year* of the period, i.e., in a set $year = [2000, 2005, 2010, 2015]$, the period '2010' comprises the years $[2006, .., 2010]$.

[5] The set `time` collects all sub-annual temporal units across all levels of temporal disaggregation. In a `MESSAGE`-scheme `ixmp.Scenario`, this set always includes an element "year", and the duration of that element is 1 ($duration\_time_{'year'} = 1$).

[6] A generic formulation of linear constraints is implemented in MESSAGE*ix*, see *Section of generic relations (linear constraints)*. These constraints can be used for testing and development, but specific new features should be implemented by specific equations and parameters.

Set name	Notation	Explanatory comments
level_resource (level)[7]	$l \in L^{RES} \subseteq L$	levels related to *fossil resources* representation
level_renewable (level)[7]	$l \in L^{REN} \subseteq L$	levels related to *renewables* representation
type_node[8]	$\widehat{n} \in \widehat{N}$	Category types for nodes
cat_node (type_node,node)	$n \in N(\widehat{n})$	Category mapping between node types and nodes
type_tec[9]	$\widehat{t} \in \widehat{T}$	Category types for technologies
cat_tec (type_tec,tec)	$t \in T(\widehat{t})$	Category mapping between tec types and technologies
inv_tec (tec)[10]	$t \in T^{INV} \subseteq T$	Specific subset of investment technologies
renewable_tec (tec)[11]	$t \in T^{REN} \subseteq T$	Specific subset of renewable-energy technologies
type_emission	$\widehat{e} \in \widehat{E}$	Category types for emissions (greenhouse gases, pollutants, etc.)
cat_emission (type_emission,emission)	$e \in E(\widehat{e})$	Category mapping between emission types and emissions
type_tec_land (type_tec)[12]	$\widehat{t} \in \widehat{T}^{LAND} \subseteq \widehat{T}$	Mapping set of technology types and land use
balance_equality (commodity,level)	$c \in C, l \in L$	Commodities and level related to *Equation COMMOD-ITY_BALANCE_LT*

Mappings sets

These sets are generated automatically when exporting a `MESSAGE`-scheme `ixmp.Scenario` to gdx using the API. They are used in the GAMS model to reduce model size by excluding non-relevant variables and equations (e.g., actitivity of a technology outside of its technical lifetime).

Set name	Notation	Explanatory comments
map_node(node,location)		mapping of nodes across hierarchy levels (location is in node)

Mapping sets (flags) for bounds

There are a number of mappings sets generated when exporting a `message_ix.Scenario` to gdx. They are used as 'flags' to indicate whether a constraint is active. The names of these sets follow the format `is_<constraint>_<dir>`.

Such mapping sets are necessary because GAMS does not distinguish between 0 and 'no value assigned', i.e., it cannot differentiate between a bound of 0 and 'no bound assigned'.

---

[7] The constraint `EXTRACTION_EQUIVALENCE` is active only for the levels included in this set, and the constraint `COMMODITY_BALANCE` is deactivated for these levels.

[8] The element "economy" is added by default as part of the `MESSAGE`-scheme `ixmp.Scenario`.

[9] The element "all" in `type_tec` and the associated mapping to all technologies in the set `cat_tec` are added by default as part of the `MESSAGE`-scheme `message_ix.Scenario`.

[10] The auxiliary set `inv_tec` (subset of `technology`) is a short-hand notation for all technologies with defined investment costs. This activates the investment cost part in the objective function and the constraints for all technologies where investment decisions are relevant. It is added by default when exporting `MESSAGE`-scheme `message_ix.Scenario` to gdx.

[11] The auxiliary set `renewable_tec` (subset of `technology`) is a short-hand notation for all technologies with defined parameters relevant for the equations in the "Renewable" section. It is added by default when exporting `MESSAGE`-scheme `message_ix.Scenario` to gdx.

[12] The mapping set `type_tec_land` is a dynamic subset of `type_tec` and specifies whether emissions from the land-use model emulator module are included when aggregating over a specific technology type. The element "all" is added by default in a `MESSAGE`-scheme `message_ix.Scenario`.

Mapping sets (flags) for fixed variables

Similar to the mapping sets for bounds, there are mapping sets to indicate whether decision variables are pre-defined to a specific value, usually taken from a solution of another model instance. This can be used to represent imperfect foresight where a policy shift or parameter change is introduced in later years. The names of these sets follow the format `is_fixed_<variable>`.

Note: This page is generated from inline documentation in `MESSAGE/parameter_def.gms`.

3.3.2 Parameter definition

This file contains the definition of all parameters used in MESSAGE*ix*.

In MESSAGE*ix*, all parameters are understood as yearly values, not as per (multi-year) period. This provides flexibility when changing the resolution of the model horizon (i.e., the set `year`).

Parameters written in *italics* are auxiliary parameters that are either generated automatically when exporting a `message_ix.Scenario` to gdx or that are computed during the pre-processing stage in GAMS.

General parameters of the MESSAGE*ix* implementation

Parameter name	Index dimensions	Explanatory comments
duration_period ($\|y\|$)[1]	`year`	duration of multi-year period (in number of years)[2]
duration_time	`time`	duration of sub-annual time slices (relative to 1)[3]
duration_time_rel	`time`|`time`	relative duration between sub-annual time slices[4]
interestrate	`year`	economy-wide interest rate or social discount rate
df_period	`year`	cumulative discount factor over period duration[4]
df_year	`year`	discount factor of the last year in the period[4]

Parameters of the *Resources* section

Parameter name	Index dimensions
resource_volume	`node`|`commodity`|`grade`
resource_cost	`node`|`commodity`|`grade`|`year`
resource_remaining	`node`|`commodity`|`grade`|`year`
bound_extraction_up	`node`|`commodity`|`level`|`year`
commodity_stock[5]	`node`|`commodity`|`level`|`year`
historical_extraction[6]	`node`|`commodity`|`grade`|`year`

---

[1] The short-hand notation $\|y\|$ is used for the parameters $duration\_period_y$ in the mathematical model documentation for exponents.

[2] The values for this parameter are computed automatically when exporting a `MESSAGE`-scheme ixmp.Scenario to gdx. Note that in MESSAGE*ix*, the elements of the `year` set are understood to be the last year in a period, see *this footnote*.

[3] The element 'year' in the set of subannual time slices `time` has the value of 1. This value is assigned by default when creating a new ixmp.Scenario based on the `MESSAGE` scheme.

[4] These parameters are computed during the GAMS execution.

[5] This parameter allows (exogenous) additions to the commodity stock over the model horizon, e.g., precipitation that replenishes the water table.

[6] Historical values of new capacity and activity can be used for parametrising the vintage structure of existing capacity and implement dynamic constraints in the first model period.

Parameters of the *Demand* section

Parameter name	Index dimensions
demand [demand_fixed][7]	`node｜commodity｜level｜year｜time`
peak_load_factor[8]	`node｜commodity｜year`

Parameters of the *Technology* section

Input/output mapping, costs and engineering specifications

Parameter name	Index names
input[9]	`node_loc ｜ tec ｜ year_vtg ｜ year_act ｜ mode ｜ node_origin ｜ commodity｜level｜time｜time_origin`
output[9]	`node_loc｜tec｜year_vtg｜year_act｜mode｜node_dest｜commodity｜level｜time｜time_dest`
inv_cost[9]	`node_loc｜tec｜year_vtg`
fix_cost[9]	`node_loc｜tec｜year_vtg｜year_act`
var_cost[9]	`node_loc｜tec｜year_vtg｜year_act｜mode｜time`
levelized_cost[10]	`node_loc｜tec｜year_vtg｜time`
construction_time	`node_loc｜tec｜year_vtg`
technical_lifetime	`node_loc｜tec｜year_vtg`
capacity_factor[9]	`node_loc｜tec｜year_vtg｜year_act｜time`
operation_factor[9]	`node_loc｜tec｜year_vtg｜year_act`
min_utilization_factor[9]	`node_loc｜tec｜year_vtg｜year_act`
rating_bin[12]	`node｜technology｜year_act｜commodity｜level｜time｜rating`
reliability_factor[8]	`node｜technology｜year_act｜commodity｜level｜time｜rating`
flexibility_factor	`node_loc ｜ technology ｜ year_vtg ｜ year_act ｜ mode ｜ commodity ｜ level｜time｜rating`
renewable_capacity_factor	`node_loc｜commodity｜grade｜level｜year`
renewable_potential	`node｜commodity｜grade｜level｜year`
emission_factor	`node_loc｜tec｜year_vtg｜year_act｜mode｜emission`

Bounds on capacity and activity

The following parameters specify upper and lower bounds on new capacity, total installed capacity, and activity.

---

[7] The parameter `demand` in a `MESSAGE`-scheme `ixmp`.Scenario is translated to the parameter `demand_fixed` in the MESSAGE implementation in GAMS. The variable `DEMAND` is introduced as an auxiliary reporting variable; it equals `demand_fixed` in a *MESSAGE*-standalone run and reports the final demand including the price response in an iterative *MESSAGE-MACRO* solution.

[8] The parameters `peak_load_factor` and `reliability_factor` are based on the formulation proposed by Sullivan et al., 2013 [6]. It is used in *Reliability of installed capacity*.

[9] Fixed and variable cost parameters and technical specifications are indexed over both the year of construction (vintage) and the year of operation (actual). This allows to represent changing technology characteristics depending on the age of the plant.

[10] The parameter `levelized_cost` is computed in the GAMS pre-processing under the assumption of full capacity utilization until the end of the technical lifetime.

[12] The upper bound of a contribution by any technology to the constraints on system reliability (*Reliability of installed capacity*) and flexibility (*Equation SYSTEM_FLEXIBILITY_CONSTRAINT*) can depend on the share of the technology output in the total commodity use at a specific level.

Parameter name	Index names
bound_new_capacity_up	node_loc|tec|year_vtg
bound_new_capacity_lo	node_loc|tec|year_vtg
bound_total_capacity_up	node_loc|tec|year_act
bound_total_capacity_lo	node_loc|tec|year_act
bound_activity_up	node_loc|tec|year_act|mode|time
bound_activity_lo	node_loc|tec|year_act|mode|time

The bounds on activity are implemented as the aggregate over all vintages in a specific period (cf. Equation `ACTIVITY_BOUND_UP` and `ACTIVITY_BOUND_LO`).

Dynamic constraints on capacity and activity

The following parameters specify constraints on the growth of new capacity and activity, i.e., market penetration.

Parameter name	Index names
initial_new_capacity_up	node_loc|tec|year_vtg
growth_new_capacity_up[13]	node_loc|tec|year_vtg
soft_new_capacity_up[13]	node_loc|tec|year_vtg
initial_new_capacity_lo	node_loc|tec|year_vtg
growth_new_capacity_lo[13]	node_loc|tec_actual|year_vtg
soft_new_capacity_lo[13]	node_loc|tec|year_vtg
initial_activity_up[14]	node_loc|tec|year_act|time
growth_activity_up[1314]	node_loc|tec|year_act|time
soft_activity_up[13]	node_loc|tec|year_act|time
initial_activity_lo[14]	node_loc|tec|year_act|time
growth_activity_lo[1314]	node_loc|tec|year_act|time
soft_activity_lo[13]	node_loc|tec|year_act|time

Parameters for the add-on technologies

The implementation of MESSAGE*ix* includes the functionality to introduce "add-on technologies" that are specifically linked to parent technologies. This feature can be used to model mitigation options (scrubber, cooling). Note, that no default addon_conversion is set, to avoid default conversion factors of 1 being set for technologies with mutiple modes, of which only a single mode should be linked to the add-on technology.

Parameter name	Index names
addon_conversion	node|tec|year_vtg|year_act|mode|time|type_addon
addon_up	node|tec|vintage|year|mode|time|type_addon
addon_lo	node|tec|vintage|year|mode|time|type_addon

The upper bound of

---

[13] All parameters related to the dynamic constraints are understood as the bound on the rate of growth/decrease, not as in percentage points and not as (1+growth rate).

[14] The dynamic constraints are not indexed over modes in the MESSAGE*ix* implementation.

Cost parameters for 'soft' relaxations of dynamic constraints

The implementation of MESSAGE*ix* includes the functionality for 'soft' relaxations of dynamic constraints on new-built capacity and activity (see Keppo and Strubegger, 2010 [3]). Refer to the section *Dynamic constraints on market penetration*.

Parameter name	Index names
abs_cost_new_capacity_soft_up	node_loc|tec|year_vtg
abs_cost_new_capacity_soft_lo	node_loc|tec|year_vtg
level_cost_new_capacity_soft_up	node_loc|tec|year_vtg
level_cost_new_capacity_soft_lo	node_loc|tec|year_vtg
abs_cost_activity_soft_up	node_loc|tec|year_act|time
abs_cost_activity_soft_lo	node_loc|tec|year_act|time
level_cost_activity_soft_up	node_loc|tec|year_act|time
level_cost_activity_soft_lo	node_loc|tec|year_act|time

Historical capacity and activity values

Historical data on new capacity and activity levels are included in MESSAGE*ix* for correct accounting of the vintage portfolio and a seamless implementation of dynamic constraints from historical years to model periods.

Parameter name	Index names
historical_new_capacity[6]	node_loc|tec|year_vtg
historical_activity[6]	node_loc|tec|year_act|mode|time

Auxiliary investment cost parameters and multipliers

Documentation not yet included.

Parameters of the *Emission* section

The implementation of MESSAGE*ix* includes a flexible and versatile accounting of emissions across different categories and species, with the option to define upper bounds and taxes on various (aggregates of) emissions and pollutants), (sets of) technologies, and (sets of) years.

Parameter name	Index dimensions
historical_emission[6]	node|emission|type_tec|year
emission_scaling[15]	type_emission|emission
bound_emission	node|type_emission|type_tec|type_year
tax_emission	node|type_emission|type_tec|type_year

Parameters of the *Land-Use model emulator* section

The implementation of MESSAGE*ix* includes a land-use model emulator, which draws on exogenous land-use scenarios (provided by another model) to derive supply of commodities (e.g., biomass) and emissions from agriculture and forestry.

---

[15] The parameters emission_scaling allows to efficiently aggregate different emissions/pollutants and set bounds or taxes on various categories.

Parameter name	Index dimensions
historical_land[6]	node|land_scenario|year
land_cost	node|land_scenario|year
land_input	node|land_scenario|year|commodity|level|time
land_output	node|land_scenario|year|commodity|level|time
land_use	node|land_scenario|year|land_type
land_emission	node|land_scenario|year|emission
initial_land_scen_up	node|land_scenario|year
growth_land_scen_up	node|land_scenario|year
initial_land_scen_lo	node|land_scenario|year
growth_land_scen_lo	node|land_scenario|year
initial_land_up	node|year|land_type
dynamic_land_up	node|land_scenario|year|land_type
growth_land_up	node|year|land_type
initial_land_lo	node|year|land_type
dynamic_land_lo	node|land_scenario|year|land_type
growth_land_lo	node|year|land_type

Parameters of the *Share Constraints* section

Share constraints define the share of a given commodity to be active on a certain level

Parameter name	Index dimensions
share_commodity_up	shares|node_share|year_act|time
share_commodity_lo	shares|node|year_act|time
share_mode_up	shares|node_loc|technology|mode|year_act|time
share_mode_lo	shares|node_loc|technology|mode|year_act|time

Parameters of the *Relations* section

Generic linear relations are implemented in MESSAGE*ix*. This feature is intended for development and testing only - all new features should be implemented as specific new mathematical formulations and associated sets & parameters.

Parameter name	Index dimensions
relation_upper	relation|node_rel|year_rel
relation_lower	relation|node_rel|year_rel
relation_cost	relation|node_rel|year_rel
relation_new_capacity	relation|node_rel|year_rel|tec
relation_total_capacity	relation|node_rel|year_rel|tec
relation_activity	relation|node_rel|year_rel|node_loc|tec|year_act|mode

Fixed variable values

The following parameters allow to set variable values to a specific value. The value is usually taken from a solution of another model instance (e.g., scenarios where a shock sets in later to mimick imperfect foresight).

The fixed values do not override any upper or lower bounds that may be defined, so fixing variables to values outside of that range will yield an infeasible model.

Parameter name	Index dimensions
fixed_extraction	`node|commodity|grade|year`
fixed_stock	`node|commodity|level|year`
fixed_new_capacity	`node|technology|year_vtg`
fixed_capacity	`node|technology|year_vtg|year_act`
fixed_activity	`node|technology|year_vtg|year_act|mode|time`
fixed_land	`node|land_scenario|year`

Note that the variable $STOCK\_CHG$ is determined implicitly by the $STOCK$ variable and therefore does not need to be explicitly fixed.

Note: This page is generated from inline documentation in `MESSAGE/model_core.gms`.

3.3.3 Mathematical formulation (core model)

The MESSAGE*ix* systems-optimization model minimizes total costs while satisfying given demand levels for commodities/services and considering a broad range of technical/engineering constraints and societal restrictions (e.g. bounds on greenhouse gas emissions, pollutants, system reliability). Demand levels are static (i.e. non-elastic), but the demand response can be integrated by linking MESSAGE*ix* to the single sector general-economy MACRO model included in this framework.

For the complete list of sets, mappings and parameters, refer to the auto-documentation pages *Sets and mappings definition* and *Parameter definition*.

Notation declaration

The following short notation is used in the mathematical description relative to the GAMS code:

Mathematical notation of sets

Math notation	GAMS set & index notation
$n \in N$	node (across spatial hierarchy levels)
$y \in Y$	year (all periods including historical and model horizon)
$y \in Y^M \subset Y$	time periods included in model horizon
$y \in Y^H \subset Y$	historical time periods (prior to first model period)
$c \in C$	commodity
$l \in L$	level
$g \in G$	grade
$t \in T$	technology (a.k.a tec)
$h \in H$	time (subannual time periods)
$m \in M$	mode
$q \in Q$	rating of non-dispatchable technologies relative to aggregate commodity use
$e \in E$	emission, pollutants
$s \in S$	scenarios of land use (for land-use model emulator)
$u \in U$	land-use types
$r \in R$	set of generic relations (linear constraints)
$t \in T^{INV} \subseteq T$	all technologies with investment decisions and capacity constraints
$t \in T^{REN} \subseteq T$	all technologies which draw their input from the renewable level
$n \in N(\widehat{n})$	all nodes that are subnodes of node $\widehat{n}$
$y \in Y(\widehat{y})$	all years mapped to the category `type_year` $\widehat{y}$
$t \in T(\widehat{t})$	all technologies mapped to the category `type_tec` $\widehat{t}$
$e \in E(\widehat{e})$	all emissions mapped to the category `type_emission` $\widehat{e}$

Decision variables

Variable	Explanatory text
$OBJ \in \mathbb{R}$	Objective value of the optimization program
$EXT_{n,c,g,y} \in \mathbb{R}_+$	Extraction of non-renewable/exhaustible resources from reserves
$STOCK_{n,c,l,y} \in \mathbb{R}_+$	Quantity in stock (storage) at start of period y
$STOCK\_CHG_{n,c,l,y,h} \in \mathbb{R}$	Input or output quantity into intertemporal commodity stock (storage)
$REN_{n,t,c,g,y,h}$	Activity of renewable technologies per grade
$CAP\_NEW_{n,t,y} \in \mathbb{R}_+$	Newly installed capacity (yearly average over period duration)
$CAP_{n,t,y^V,y} \in \mathbb{R}_+$	Maintained capacity in year y of vintage y^V
$CAP\_FIRM_{n,t,c,l,y,q}$	Capacity counting towards firm (dispatchable)
$ACT_{n,t,y^V,y,m,h} \in \mathbb{R}$	Activity of a technology (by vintage, mode, subannual time)
$ACT\_RATING_{n,t,y^V,y,c,l,h,q}$	Activity attributed to a particular rating bin[1]
$CAP\_NEW\_UP_{n,t,y} \in \mathbb{R}_+$	Relaxation of upper dynamic constraint on new capacity
$CAP\_NEW\_LO_{n,t,y} \in \mathbb{R}_+$	Relaxation of lower dynamic constraint on new capacity
$ACT\_UP_{n,t,y,h} \in \mathbb{R}_+$	Relaxation of upper dynamic constraint on activity[2]
$ACT\_LO_{n,t,y,h} \in \mathbb{R}_+$	Relaxation of lower dynamic constraint on activity[2]
$LAND_{n,s,y} \in [0,1]$	Relative share of land-use scenario (for land-use model emulator)
$EMISS_{n,e,\widehat{t},y}$	Auxiliary variable for aggregate emissions by technology type
$REL_{r,n,y} \in \mathbb{R}$	Auxiliary variable for left-hand side of relations (linear constraints)
$COMMODITY\_USE_{n,c,l,y}$	Auxiliary variable for amount of commodity used at specific level

---

[1] The auxiliary variable $ACT\_RATING_{n,t,y^V,y,c,l,h,q}$ is defined in terms of input or output of the technology.

[2] The dynamic activity constraints are implemented as summed over all modes; therefore, the variables for the relaxation are not indexed over the set `mode`.

The index y^V is the year of construction (vintage) wherever it is necessary to clearly distinguish between year of construction and the year of operation.

All decision variables are by year, not by (multi-year) period, except $STOCK_{n,c,l,y}$. In particular, the new capacity variable $CAP\_NEW_{n,t,y}$ has to be multiplied by the number of years in a period $|y| = duration\_period_y$ to determine the available capacity in subsequent periods. This formulation gives more flexibility when it comes to using periods of different duration (more intuitive comparison across different periods).

The current model framework allows both input or output normalized formulation. This will affect the parametrization, see Section *Efficiency - output- vs. input defined technologies* for more details.

Auxiliary variables

Variable	Explanatory text
$DEMAND_{n,c,l,y,h} \in \mathbb{R}$	Demand level (in equilibrium with MACRO integration)
$PRICE\_COMMODITY_{n,c,l,y,h}$	Commodity price (undiscounted marginals of the commodity balances)
$PRICE\_EMISSION_{n,e,\widehat{t},y}$	Emission price (undiscounted marginals of EMISSION_BOUND constraint)
$COST\_NODAL\_NET_{n,y} \in \mathbb{R}$	System costs at the node level net of energy trade revenues/cost
$GDP_{n,y} \in \mathbb{R}$	gross domestic product (GDP) in market exchange rates for MACRO reporting

Objective function

The objective function of the MESSAGE*ix* core model

Equation OBJECTIVE

The objective function (of the core model) minimizes total discounted systems costs including costs for emissions, relaxations of dynamic constraints

$$OBJ = \sum_{n,y \in YM} df\_year_y \cdot COST\_NODAL_{n,y}$$

Regional system cost accounting function

Accounting of regional system costs over time

Equation COST_ACCOUNTING_NODAL

Accounting of regional systems costs over time as well as costs for emissions (taxes), land use (from the model land-use model emulator), relaxations of dynamic constraints, and linear relations.

$$COST\_NODAL_{n,y} = \sum_{c,g} resource\_cost_{n,c,g,y} \cdot EXT_{n,c,g,y}$$

$$+ \sum_{t} \Bigg(inv\_cost_{n,t,y} \cdot construction\_time\_factor_{n,t,y}$$

$$\cdot end\_of\_horizon\_factor_{n,t,y} \cdot CAP\_NEW_{n,t,y}$$

$$+ \sum_{y^V \leq y} fix\_cost_{n,t,y^V,y} \cdot CAP_{n,t,y^V,y}$$

$$+ \sum_{\substack{y^V \leq y \\ m,h}} var\_cost_{n,t,y^V,y,m,h} \cdot ACT_{n,t,y^V,y,m,h}$$

$$+ \Big(abs\_cost\_new\_capacity\_soft\_up_{n,t,y}$$

$$+ level\_cost\_new\_capacity\_soft\_up_{n,t,y} \cdot inv\_cost_{n,t,y} \Big) \cdot CAP\_NEW\_UP_{n,t,y}$$

$$+ \Big(abs\_cost\_new\_capacity\_soft\_lo_{n,t,y}$$

$$+ level\_cost\_new\_capacity\_soft\_lo_{n,t,y} \cdot inv\_cost_{n,t,y} \Big) \cdot CAP\_NEW\_LO_{n,t,y}$$

$$+ \sum_{m,h} \Big(abs\_cost\_activity\_soft\_up_{n,t,y,m,h}$$

$$+ level\_cost\_activity\_soft\_up_{n,t,y,m,h} \cdot levelized\_cost_{n,t,y,m,h} \Big) \cdot ACT\_UP_{n,t,y,h}$$

$$+ \sum_{m,h} \Big(abs\_cost\_activity\_soft\_lo_{n,t,y,m,h}$$

$$+ level\_cost\_activity\_soft\_lo_{n,t,y,m,h} \cdot levelized\_cost_{n,t,y,m,h} \Big) \cdot ACT\_LO_{n,t,y,h} \Bigg)$$

$$+ \sum_{\substack{\widehat{e},\widehat{t} \\ e \in E(\widehat{e})}} emission\_scaling_{\widehat{e},e} \cdot emission\_tax_{n,\widehat{e},\widehat{t},y} \cdot EMISS_{n,e,\widehat{t},y}$$

$$+ \sum_{s} land\_cost_{n,s,y} \cdot LAND_{n,s,y}$$

$$+ \sum_{r} relation\_cost_{r,n,y} \cdot REL_{r,n,y}$$

Here, $n^L \in N(n)$ are all nodes n^L that are sub-nodes of node n. The subset of technologies $t \in T(\widehat{t})$ are all tecs that belong to category $\widehat{t}$, and similar notation is used for emissions $e \in E$.

Resource and commodity section

Constraints on resource extraction

Equation EXTRACTION_EQUIVALENCE

This constraint translates the quantity of resources extracted (summed over all grades) to the input used by all technologies (drawing from that node). It is introduced to simplify subsequent notation in input/output relations and nodal

balance constraints.

$$\sum_g EXT_{n,c,g,y} = \sum_{\substack{n^L,t,m,h,h^{OD} \\ y^V \le y \\ l \in L^{RES} \subseteq L}} input_{n^L,t,y^V,y,m,n,c,l,h,h^{OD}} \cdot ACT_{n^L,t,m,y,h}$$

The set $L^{RES} \subseteq L$ denotes all levels for which the detailed representation of resources applies.

Equation EXTRACTION_BOUND_UP

This constraint specifies an upper bound on resource extraction by grade.

$$EXT_{n,c,g,y} \le bound\_extraction\_up_{n,c,g,y}$$

Equation RESOURCE_CONSTRAINT

This constraint restricts that resource extraction in a year guarantees the "remaining resources" constraint, i.e., only a given fraction of remaining resources can be extracted per year.

$$EXT_{n,c,g,y} \le resource\_remaining_{n,c,g,y} \cdot \Big(resource\_volume_{n,c,g}$$
$$- \sum_{y'<y} duration\_period_{y'} \cdot EXT_{n,c,g,y'}\Big)$$

Equation RESOURCE_HORIZON

This constraint ensures that total resource extraction over the model horizon does not exceed the available resources.

$$\sum_y duration\_period_y \cdot EXT_{n,c,g,y} \le resource\_volume_{n,c,g}$$

Constraints on commodities and stocks

Auxiliary COMMODITY_BALANCE

For the commodity balance constraints below, we introduce an auxiliary *COMMODITY_BALANCE*. This is implemented as a GAMS *$macro* function.

$$\sum_{\substack{n^L,t,m,h^A \\ y^V \leq y}} output_{n^L,t,y^V,y,m,n,c,l,h^A,h} \cdot duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}$$

$$- \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y}} input_{n^L,t,y^V,y,m,n,c,l,h^A,h} \cdot duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,m,y,h^A}$$

$$+ STOCK\_CHG_{n,c,l,y,h}$$

$$+ \sum_s \Big(land\_output_{n,s,y,c,l,h} - land\_input_{n,s,y,c,l,h}\Big) \cdot LAND_{n,s,y}$$

$$- demand\_fixed_{n,c,l,y,h} = COMMODITY\_BALANCE_{n,c,l,y,h} \quad \forall\, l \notin (L^{RES}$$

The commodity balance constraint at the resource level is included in the *Equation RESOURCE_CONSTRAINT*, while at the renewable level, it is included in the *Equation RENEWABLES_EQUIVALENCE*.

Equation COMMODITY_BALANCE_GT

This constraint ensures that supply is greater or equal than demand for every commodity-level combination.

$$COMMODITY\_BALANCE_{n,c,l,y,h} \geq 0$$

Equation COMMODITY_BALANCE_LT

This constraint ensures the supply is smaller than or equal to the demand for all commodity-level combinatio given in the $balance\_equality_{c,l}$. In combination withe constraint above, it ensures that supply is (exactly) equal to demand.

$$COMMODITY\_BALANCE_{n,c,l,y,h} \leq 0$$

Equation STOCKS_BALANCE

This constraint ensures the inter-temporal balance of commodity stocks. The parameter $commodity\_stocks_{n,c,l}$ can be used to model exogenous additions to the stock

$$STOCK_{n,c,l,y} + commodity\_stock_{n,c,l,y} = duration\_period_y \cdot \sum_h STOCK\_CHG_{n,c,l,y,h}$$
$$+ STOCK_{n,c,l,y+1}$$

Technology section

Technical and engineering constraints

The first set of constraints concern technologies that have explicit investment decisions and where installed/maintained capacity is relevant for operational decisions. The set where $T^{INV} \subseteq T$ is the set of all these technologies.

Equation CAPACITY_CONSTRAINT

This constraint ensures that the actual activity of a technology at a node cannot exceed available (maintained) capacity summed over all vintages, including the technology capacity factor $capacity\_factor_{n,t,y,t}$.

$$\sum_m ACT_{n,t,y^V,y,m,h} \leq duration\_time_h \cdot capacity\_factor_{n,t,y^V,y,h} \cdot CAP_{n,t,y^V,y} \quad \forall\, t\, \in\, T^{INV}$$

Equation CAPACITY_MAINTENANCE_HIST

The following three constraints implement technology capacity maintenance over time to allow early retirment. The optimization problem determines the optimal timing of retirement, when fixed operation-and-maintenance costs exceed the benefit in the objective function.

The first constraint ensures that historical capacity (built prior to the model horizon) is available as installed capacity in the first model period.

$$CAP_{n,t,y^V,'first\_period'} \leq remaining\_capacity_{n,t,y^V,'first\_period'} \cdot duration\_period_{y^V} \cdot historical\_new\_capacity_{n,t,y^V}$$
$$\text{if } y^V <' first\_period' \text{ and } |y| - |y^V| < technical\_lifetime_{n,t,y^V} \quad \forall\, t \in T^{INV}$$

Equation CAPACITY_MAINTENANCE_NEW

The second constraint ensures that capacity is fully maintained throughout the model period in which it was constructed (no early retirement in the period of construction).

$$CAP_{n,t,y^V,y^V} = remaining\_capacity_{n,t,y^V,y^V} \cdot duration\_period_{y^V} \cdot CAP\_NEW n, t, y^V \quad \forall\, t \in T^{INV}$$

The current formulation does not account for construction time in the constraints, but only adds a mark-up to the investment costs in the objective function.

Equation CAPACITY_MAINTENANCE

The third constraint implements the dynamics of capacity maintenance throughout the model horizon. Installed capacity can be maintained over time until decommissioning, which is irreversible.

$$CAP_{n,t,y^V,y} \leq remaining\_capacity_{n,t,y^V,y} \cdot CAP_{n,t,y^V,y-1}$$
$$\text{if } y > y^V \text{ and } y^V >' first\_period' \text{ and } |y| - |y^V| < technical\_lifetime_{n,t,y^V} \quad \forall\, t \in T^{INV}$$

Equation OPERATION_CONSTRAINT

This constraint provides an upper bound on the total operation of installed capacity over a year. It can be used to represent reuqired scheduled unavailability of installed capacity.

$$\sum_{m,h} ACT_{n,t,y^V,y,m,h} \leq operation\_factor_{n,t,y^V,y} \cdot capacity\_factor_{n,t,y^V,y,m,\text{'year'}} \cdot CAP_{n,t,y^V,y} \quad \forall\, t \in T^{INV}$$

This constraint is only active if $operation\_factor_{n,t,y^V,y} < 1$.

Equation MIN_UTILIZATION_CONSTRAINT

This constraint provides a lower bound on the total utilization of installed capacity over a year.

$$\sum_{m,h} ACT_{n,t,y^V,y,m,h} \geq min\_utilization\_factor_{n,t,y^V,y} \cdot CAP_{n,t,y^V,y} \quad \forall\, t \in T^{INV}$$

This constraint is only active if $min\_utilization\_factor_{n,t,y^V,y}$ is defined.

Constraints representing renewable integration

Equation RENEWABLES_EQUIVALENCE

This constraint defines the auxiliary variables REN to be equal to the output of renewable technologies (summed over grades).

$$\sum_{g} REN_{n,t,c,g,y,h} \leq \sum_{\substack{n,t,m,l,h,h^{OD} \\ y^V \leq y \\ l \in L^{REN} \subseteq L}} input_{n^L,t,y^V,y,m,n,c,l,h,h^{OD}} \cdot ACT_{n^L,t,m,y,h}$$

The set $L^{REN} \subseteq L$ denotes all levels for which the detailed representation of renewables applies.

Equation RENEWABLES_POTENTIAL_CONSTRAINT

This constraint sets the potential potential by grade as the upper bound for the auxiliary variable REN.

$$\sum_{\substack{t,h \\ t \in T^R \subseteq t}} REN_{n,t,c,g,y,h} \leq \sum_{\substack{l \\ l \in L^R \subseteq L}} renewable\_potential_{n,c,g,l,y}$$

Equation RENEWABLES_CAPACITY_REQUIREMENT

This constraint connects the capacity factor of a renewable grade to the installed capacity of a technology. It sets the lower limit for the capacity of a renewable technology to the summed activity over all grades (REN) devided by the capactiy factor of this grade. It represents the fact that different renewable grades require different installed capacities to provide their full potential.

$$\sum_{y^V, h} CAP_{n,t,y^V,y} \cdot operation\_factor_{n,t,y^V,y} \cdot capacity\_factor_{n,t,y^V,y,h}$$

$$\geq \sum_{g,h,l} \frac{1}{renewable\_capacity\_factor_{n,c,g,l,y}} \cdot REN_{n,t,c,g,y,h}$$

This constraint is only active if $renewable\_capacity\_factor_{n,c,g,l,y}$ is defined.

Constraints for addon technologies

Equation ADDON_ACTIVITY_UP

This constraint provides an upper bound on the activity of an addon technology that can only be operated jointly with a parent technology (e.g., abatement option, SO2 scrubber, power plant cooling technology).

$$\sum_{t' \sim t^A, y^V \leq y} ACT_{n,t',y^V,y,m,h} \leq \sum_{t,y^V \leq y} addon\_up_{n,t^a,y,m,h,t^A} \cdot addon\_conversion_{n,t',y^V,y,m,h} \cdot ACT_{n,t,y^V,y,m,h}$$

Equation ADDON_ACTIVITY_LO

This constraint provides a lower bound on the activity of an addon technology that has to be operated jointly with a parent technology (e.g., power plant cooling technology). The parameter *addon_lo* allows to define a minimum level of operation of addon technologies relative to the activity of the parent technology. If *addon_minimum = 1*, this means that it is mandatory to operate the addon technology at the same level as the parent technology (i.e., full mitigation).

$$\sum_{t' \sim t^A, y^V \leq y} ACT_{n,t',y^V,y,m,h} \geq \sum_{t,y^V \leq y} addon\_lo_{n,t^a,y,m,h,t^A} \cdot addon\_conversion_{n,t',y^V,y,m,h} \cdot ACT_{n,t,y^V,y,m,h}$$

System reliability and flexibility requirements

This section followi allows to include system-wide reliability and flexility considerations. The current formulation is based on Sullivan et al., 2013 [6].

Aggregate use of a commodity

The system reliability and flexibility constraints are implemented using an auxiliary variable representing the total use (i.e., input of each commodity per level).

Equation COMMODITY_USE_LEVEL

This constraint defines the auxiliary variable $COMMODITY\_USE_{n,c,l,y}$, which is used to define the rating bins and the peak-load that needs to be offset with firm (dispatchable) capacity.

$$COMMODITY\_USE_{n,c,l,y} = \sum_{n^L,t,y^V,m,h} input_{n^L,t,y^V,y,m,n,c,l,h,h}$$
$$\cdot \, duration\_time\_rel_{h,h} \cdot ACT_{n^L,t,y^V,y,m,h}$$

This constraint and the auxiliary variable is only active if $peak\_load\_factor_{n,c,l,y,h}$ or $flexibility\_factor_{n,t,y^V,y,m,c,l,h,r}$ is defined.

Auxilary variables for technology activity by "rating bins"

The capacity and activity of certain (usually non-dispatchable) technologies can be assumed to only partially contribute to the system reliability and flexibility requirements.

Equation ACTIVITY_RATING_BIN

The auxiliary variable for rating-specific activity of each technology cannot exceed the share of the rating bin in relation to the total commodity use.

$$ACT\_RATING_{n,t,y^V,y,c,l,h,q} \leq rating\_bin_{n,t,y,c,l,h,q} \cdot COMMODITY\_USE_{n,c,l,y}$$

Equation ACTIVITY_SHARE_TOTAL

The sum of the auxiliary rating-specific activity variables need to equal the total input and/or output of the technology.

$$\sum_q ACT\_RATING_{n,t,y^V,y,c,l,h,q} = \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y}} (input_{n^L,t,y^V,y,m,n,c,l,h^A,h} + output_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$
$$\cdot \, duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}$$

Reliability of installed capacity

The "firm capacity" that a technology can contribute to system reliability depends on its dispatch characteristics. For dispatchable technologies, the total installed capacity counts toward the firm capacity constraint. This is active if the parameter is defined over $reliability\_factor_{n,t,y,c,l,h,'firm'}$. For non-dispatchable technologies, or those that do not have explicit investment decisions, the contribution to system reliability is calculated by using the auxiliary variable $ACT\_RATING_{n,t,y^V,y,c,l,h,q}$ as a proxy, with the $reliability\_factor_{n,t,y,c,l,h,q}$ defined per rating bin q.

Equation FIRM_CAPACITY_PROVISION

Technologies where the reliability factor is defined with the rating *firm* have an auxiliary variable $CAP\_FIRM_{n,t,c,l,y,q}$, defined in terms of output.

$$\sum_q CAP\_FIRM_{n,t,c,l,y,q} = \sum_{y^V \leq y} output_{n^L,t,y^V,y,m,n,c,l,h^A,h} \cdot duration\_time_h$$
$$\cdot \, capacity\_factor_{n,t,y^V,y,h} \cdot CAP_{n,t,y^Y,y} \quad \forall \, t \in T^{INV}$$

Equation SYSTEM_RELIABILITY_CONSTRAINT

This constraint ensures that there is sufficient firm (dispatchable) capacity in each period. The formulation is based on Sullivan et al., 2013 [6].

$$\sum_{\substack{t,q \\ y^V \leq y}}^{t \in T^{INV}} reliability\_factor_{n,t,y,c,l,h,'firm'} \cdot CAP\_FIRM_{n,t,c,l,y}$$

$$+ \sum_{t,q,y^V \leq y} reliability\_factor_{n,t,y,c,l,h,q} \cdot ACT\_RATING_{n,t,y^V,y,c,l,h,q}$$

$$\geq peak\_load\_factor_{n,c,l,y,h} \cdot COMMODITY\_USE_{n,c,l,y}$$

This constraint is only active if $peak\_load\_factor_{n,c,l,y,h}$ is defined.

Equation SYSTEM_FLEXIBILITY_CONSTRAINT

This constraint ensures that, in each sub-annual time slice, there is a sufficient contribution from flexible technologies to ensure smooth system operation.

$$\sum_{\substack{n^L,t,m,h^A \\ y^V \leq y}} flexibility\_factor_{n^L,t,y^V,y,m,c,l,h,'unrated'}$$

$$\cdot (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$
$$\cdot duration\_time\_rel_{h,h^A} \cdot ACT_{n,t,y^V,y,m,h}$$
$$+ \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y}} flexibility\_factor_{n^L,t,y^V,y,m,c,l,h,1}$$

$$\cdot (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$
$$\cdot duration\_time\_rel_{h,h^A} \cdot ACT\_RATING_{n,t,y^V,y,c,l,h,q} \geq 0$$

Bounds on capacity and activity

Equation NEW_CAPACITY_BOUND_UP

This constraint provides upper bounds on new capacity installation.

$$CAP\_NEW_{n,t,y} \leq bound\_new\_capacity\_up_{n,t,y} \quad \forall \, t \, \in \, T^{INV}$$

Equation NEW_CAPACITY_BOUND_LO

This constraint provides lower bounds on new capacity installation.

$$CAP\_NEW_{n,t,y} \geq bound\_new\_capacity\_lo_{n,t,y} \quad \forall \, t \, \in \, T^{INV}$$

Equation TOTAL_CAPACITY_BOUND_UP

This constraint gives upper bounds on the total installed capacity of a technology in a specific year of operation summed over all vintages.

$$\sum_{y^V \leq y} CAP_{n,t,y,y^V} \leq bound\_total\_capacity\_up_{n,t,y} \quad \forall\, t \, \in \, T^{INV}$$

Equation TOTAL_CAPACITY_BOUND_LO

This constraint gives lower bounds on the total installed capacity of a technology.

$$\sum_{y^V \leq y} CAP_{n,t,y,y^V} \geq bound\_total\_capacity\_lo_{n,t,y} \quad \forall\, t \, \in \, T^{INV}$$

Equation ACTIVITY_BOUND_UP

This constraint provides upper bounds by mode of a technology activity, summed over all vintages.

$$\sum_{y^V \leq y} ACT_{n,t,y^V,y,m,h} \leq bound\_activity\_up_{n,t,m,y,h}$$

Equation ACTIVITY_BOUND_ALL_MODES_UP

This constraint provides upper bounds of a technology activity across all modes and vintages.

$$\sum_{y^V \leq y,m} ACT_{n,t,y^V,y,m,h} \leq bound\_activity\_up_{n,t,y,'all',h}$$

Equation ACTIVITY_BOUND_LO

This constraint provides lower bounds by mode of a technology activity, summed over all vintages.

$$\sum_{y^V \leq y} ACT_{n,t,y^V,y,m,h} \geq bound\_activity\_lo_{n,t,y,m,h}$$

We assume that $bound\_activity\_lo_{n,t,y,m,h} = 0$ unless explicitly stated otherwise.

Equation ACTIVITY_BOUND_ALL_MODES_LO

This constraint provides lower bounds of a technology activity across all modes and vintages.

$$\sum_{y^V \leq y, m} ACT_{n,t,y^V,y,m,h} \geq bound\_activity\_lo_{n,t,y,'all',h}$$

We assume that $bound\_activity\_lo_{n,t,y,'all',h} = 0$ unless explicitly stated otherwise.

Constraints on shares of technologies and commodities

This section allows to include upper and lower bounds on the shares of modes used by a technology or the shares of commodities produced or consumed by groups of technologies.

Share constraints on activity by mode

Equation SHARES_MODE_UP

This constraint provides upper bounds of the share of the activity of one mode of a technology. For example, it could limit the share of heat that can be produced in a combined heat and electricity power plant.

$$ACT_{n^L,t,y^V,y,m,h^A} \leq share\_mode\_up_{s,n,y,m,h} \cdot \sum_{m\prime} ACT_{n^L,t,y^V,y,m\prime,h^A}$$

Equation SHARES_MODE_LO

This constraint provides lower bounds of the share of the activity of one mode of a technology.

$$ACT_{n^L,t,y^V,y,m,h^A} \geq share\_mode\_lo_{s,n,y,m,h} \cdot \sum_{m\prime} ACT_{n^L,t,y^V,y,m\prime,h^A}$$

Share constraints on commodities

These constraints allow to set upper and lower bound on the quantity of commodities produced or consumed by a group of technologies relative to the commodities produced or consumed by another group.

The implementation is generic and flexible, so that any combination of commodities, levels, technologies and nodes can be put in relation to any other combination.

The notation S^{share} represents the mapping set *map_shares_commodity_share* denoting all technology types, nodes, commodities and levels to be included in the numerator, and S^{total} is the equivalent mapping set *map_shares_commodity_total* for the denominator.

Equation SHARE_CONSTRAINT_COMMODITY_UP

$$\sum_{\substack{n^L,t,m,h^A \\ y^V \leq y,(n,\hat{t},m,c,l) \sim S^{share}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$

$$\cdot\, duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}$$

$$\geq share\_commodity\_up_{s,n,y,h} \cdot \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y,(n,\hat{t},m,c,l) \sim S^{total}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$

$$\cdot\, duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}$$

This constraint is only active if $share\_commodity\_up_{s,n,y,h}$ is defined.

Equation SHARE_CONSTRAINT_COMMODITY_LO

$$\sum_{\substack{n^L,t,m,h^A \\ y^V \leq y,(n,\hat{t},m,c,l) \sim S^{share}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$

$$\cdot\, duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}$$

$$\leq share\_commodity\_lo_{s,n,y,h} \cdot \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y,(n,\hat{t},m,c,l) \sim S^{total}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h})$$

$$\cdot\, duration\_time\_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}$$

This constraint is only active if $share\_commodity\_lo_{s,n,y,h}$ is defined.

Dynamic constraints on market penetration

The constraints in this section specify dynamic upper and lower bounds on new capacity and activity, i.e., constraints on market penetration and rate of expansion or phase-out of a technology.

The formulation directly includes the option for 'soft' relaxations of dynamic constraints (cf. Keppo and Strubegger, 2010 [3]).

Equation NEW_CAPACITY_CONSTRAINT_UP

The level of new capacity additions cannot be greater than an initial value (compounded over the period duration), annual growth of the existing 'capital stock', and a "soft" relaxation of the upper bound.

$$CAP\_NEW_{n,t,y} \leq initial\_new\_capacity\_up_{n,t,y} \cdot \frac{\left(1 + growth\_new\_capacity\_up_{n,t,y}\right)^{|y|} - 1}{growth\_new\_capacity\_up_{n,t,y}}$$
$$+ \left(CAP\_NEW_{n,t,y-1} + historical\_new\_capacity_{n,t,y-1}\right)$$
$$\cdot \left(1 + growth\_new\_capacity\_up_{n,t,y}\right)^{|y|}$$
$$+ CAP\_NEW\_UP_{n,t,y} \cdot \left(\left(1 + soft\_new\_capacity\_up_{n,t,y}\right)^{|y|} - 1\right)$$
$$\forall\, t\, \in\, T^{INV}$$

Here, $|y|$ is the number of years in period y, i.e., $duration\_period_y$.

Equation NEW_CAPACITY_SOFT_CONSTRAINT_UP

This constraint ensures that the relaxation of the dynamic constraint on new capacity (investment) does not exceed the level of the investment in the same period (cf. Keppo and Strubegger, 2010 [3]).

$$CAP\_NEW\_UP_{n,t,y} \leq CAP\_NEW_{n,t,y} \quad \forall\, t\, \in\, T^{INV}$$

Equation NEW_CAPACITY_CONSTRAINT_LO

This constraint gives dynamic lower bounds on new capacity.

$$CAP\_NEW_{n,t,y} \geq - initial\_new\_capacity\_lo_{n,t,y} \cdot \frac{\left(1 + growth\_new\_capacity\_lo_{n,t,y}\right)^{|y|}}{growth\_new\_capacity\_lo_{n,t,y}}$$
$$+ \left(CAP\_NEW_{n,t,y-1} + historical\_new\_capacity_{n,t,y-1}\right)$$
$$\cdot \left(1 + growth\_new\_capacity\_lo_{n,t,y}\right)^{|y|}$$
$$- CAP\_NEW\_LO_{n,t,y} \cdot \left(\left(1 + soft\_new\_capacity\_lo_{n,t,y}\right)^{|y|} - 1\right)$$
$$\forall\, t\, \in\, T^{INV}$$

Equation NEW_CAPACITY_SOFT_CONSTRAINT_LO

This constraint ensures that the relaxation of the dynamic constraint on new capacity does not exceed level of the investment in the same year.

$$CAP\_NEW\_LO_{n,t,y} \leq CAP\_NEW_{n,t,y} \quad \forall\, t\, \in\, T^{INV}$$

Equation ACTIVITY_CONSTRAINT_UP

This constraint gives dynamic upper bounds on the market penetration of a technology activity.

$$\sum_{y^V \leq y, m} ACT_{n,t,y^V,y,m,h} \leq initial\_activity\_up_{n,t,y,h} \cdot \frac{\left(1 + growth\_activity\_up_{n,t,y,h}\right)^{|y|} - 1}{growth\_activity\_up_{n,t,y,h}}$$

$$+ \left(\sum_{y^V \leq y-1, m} ACT_{n,t,y^V,y-1,m,h} + \sum_m historical\_activity_{n,t,y-1,m,h} \right)$$

$$\cdot \left(1 + growth\_activity\_up_{n,t,y,h}\right)^{|y|}$$

$$+ ACT\_UP_{n,t,y,h} \cdot \left(\left(1 + soft\_activity\_up_{n,t,y,h}\right)^{|y|} - 1 \right)$$

Equation ACTIVITY_SOFT_CONSTRAINT_UP

This constraint ensures that the relaxation of the dynamic activity constraint does not exceed the level of the activity.

$$ACT\_UP_{n,t,y,h} \leq \sum_{y^V \leq y, m} ACT_{n^L,t,y^V,y,m,h}$$

Equation ACTIVITY_CONSTRAINT_LO

This constraint gives dynamic lower bounds on the market penetration of a technology activity.

$$\sum_{y^V \leq y, m} ACT_{n,t,y^V,y,m,h} \geq - initial\_activity\_lo_{n,t,y,h} \cdot \frac{\left(1 + growth\_activity\_lo_{n,t,y,h}\right)^{|y|} - 1}{growth\_activity\_lo_{n,t,y,h}}$$

$$+ \left(\sum_{y^V \leq y-1, m} ACT_{n,t,y^V,y-1,m,h} + \sum_m historical\_activity_{n,t,y-1,m,h} \right)$$

$$\cdot \left(1 + growth\_activity\_lo_{n,t,y,h}\right)^{|y|}$$

$$- ACT\_LO_{n,t,y,h} \cdot \left(\left(1 + soft\_activity\_lo_{n,t,y,h}\right)^{|y|} - 1 \right)$$

Equation ACTIVITY_SOFT_CONSTRAINT_LO

This constraint ensures that the relaxation of the dynamic activity constraint does not exceed the level of the activity.

$$ACT\_LO_{n,t,y,h} \leq \sum_{y^V \leq y, m} ACT_{n,t,y^V,y,m,h}$$

Emission section

Auxiliary variable for aggregate emissions

Equation EMISSION_EQUIVALENCE

This constraint simplifies the notation of emissions aggregated over different technology types and the land-use model emulator. The formulation includes emissions from all sub-nodes n^L of n.

$$
EMISS_{n,e,\widehat{t},y} = \sum_{n^L \in N(n)} \left(\sum_{t \in T(\widehat{t}), y^V \leq y, m, h} emission\_factor_{n^L,t,y^V,y,m,e} \cdot ACT_{n^L,t,y^V,y,m,h} \right.
$$
$$
\left. + \sum_{s} land\_emission_{n^L,s,y,e} \cdot LAND_{n^L,s,y} \text{ if } \widehat{t} \in \widehat{T}^{LAND} \right)
$$

Bound on emissions

Equation EMISSION_CONSTRAINT

This constraint enforces upper bounds on emissions (by emission type). For all bounds that include multiple periods, the parameter $bound\_emission_{n,\widehat{e},\widehat{t},\widehat{y}}$ is scaled to represent average annual emissions over all years included in the year-set $\widehat{y}$.

The formulation includes historical emissions and allows to model constraints ranging over both the model horizon and historical periods.

$$
\frac{\sum_{y' \in Y(\widehat{y}), e \in E(\widehat{e})} \begin{array}{c} duration\_period_{y'} \cdot emission\_scaling_{\widehat{e},e} \cdot \\ \left(EMISS_{n,e,\widehat{t},y'} + \sum_{m} historical\_emission_{n,e,\widehat{t},y'} \right) \end{array}}{\sum_{y' \in Y(\widehat{y})} duration\_period_{y'}} \leq bound\_emission_{n,\widehat{e},\widehat{t},\widehat{y}}
$$

Land-use model emulator section

Bounds on total land use

Equation LAND_CONSTRAINT

This constraint enforces a meaningful result of the land-use model emulator, in particular a bound on the total land used in MESSAGE*ix*. The linear combination of land scenarios must be equal to 1.

$$
\sum_{s \in S} LAND_{n,s,y} = 1
$$

Dynamic constraints on land use

These constraints enforces upper and lower bounds on the change rate per land scenario.

Equation DYNAMIC_LAND_SCEN_CONSTRAINT_UP

$$LAND_{n,s,y} \leq initial\_land\_scen\_up_{n,s,y} \cdot \frac{\left(1 + growth\_land\_scen\_up_{n,s,y}\right)^{|y|} - 1}{growth\_land\_scen\_up_{n,s,y}}$$
$$+ \left(LAND_{n,s,y-1} + historical\_land_{n,s,y-1}\right) \cdot \left(1 + growth\_land\_scen\_up_{n,s,y}\right)^{|y|}$$

Equation DYNAMIC_LAND_SCEN_CONSTRAINT_LO

$$LAND_{n,s,y} \geq - initial\_land\_scen\_lo_{n,s,y} \cdot \frac{\left(1 + growth\_land\_scen\_lo_{n,s,y}\right)^{|y|} - 1}{growth\_land\_scen\_lo_{n,s,y}}$$
$$+ \left(LAND_{n,s,y-1} + historical\_land_{n,s,y-1}\right) \cdot \left(1 + growth\_land\_scen\_lo_{n,s,y}\right)^{|y|}$$

These constraints enforces upper and lower bounds on the change rate per land type determined as a linear combination of land use scenarios.

Equation DYNAMIC_LAND_TYPE_CONSTRAINT_UP

$$\sum_{s \in S} land\_use_{n,s,y,u} \cdot LAND_{n,s,y} \leq initial\_land\_up_{n,y,u} \cdot \frac{\left(1 + growth\_land\_up_{n,y,u}\right)^{|y|} - 1}{growth\_land\_up_{n,y,u}}$$
$$+ \left(\sum_{s \in S} \left(land\_use_{n,s,y-1,u} + dynamic\_land\_up_{n,s,y-1,u}\right) \right.$$
$$\cdot \left(LAND_{n,s,y-1} + historical\_land_{n,s,y-1}\right) \Big)$$
$$\cdot \left(1 + growth\_land\_up_{n,y,u}\right)^{|y|}$$

Equation DYNAMIC_LAND_TYPE_CONSTRAINT_LO

$$\sum_{s \in S} land\_use_{n,s,y,u} \cdot LAND_{n,s,y} \geq -initial\_land\_lo_{n,y,u} \cdot \frac{\left(1 + growth\_land\_lo_{n,y,u}\right)^{|y|} - 1}{growth\_land\_lo_{n,y,u}}$$
$$+ \left(\sum_{s \in S} \left(land\_use_{n,s,y-1,u} + dynamic\_land\_lo_{n,s,y-1,u}\right) \right.$$
$$\cdot \left(LAND_{n,s,y-1} + historical\_land_{n,s,y-1}\right) \Big)$$
$$\cdot \left(1 + growth\_land\_lo_{n,y,u}\right)^{|y|}$$

Section of generic relations (linear constraints)

This feature is intended for development and testing only - all new features should be implemented as specific new mathematical formulations and associated sets & parameters!

Auxiliary variable for left-hand side

Equation RELATION_EQUIVALENCE

$$
\begin{aligned}
REL_{r,n,y} = \sum_t \Bigg(& relation\_new\_capacity_{r,n,y,t} \cdot CAP\_NEW_{n,t,y} \\
& + relation\_total\_capacity_{r,n,y,t} \cdot \sum_{y^V \leq y} CAP_{n,t,y^V,y} \\
& + \sum_{n^L,y',m,h} relation\_activity_{r,n,y,n^L,t,y',m} \\
& \cdot \Big(\sum_{y^V \leq y'} ACT_{n^L,t,y^V,y',m,h} + historical\_activity_{n^L,t,y',m,h} \Big) \Bigg)
\end{aligned}
$$

The parameter $historical\_new\_capacity_{r,n,y}$ is not included here, because relations can only be active in periods included in the model horizon and there is no "writing" of capacity relation factors across periods.

Upper and lower bounds on user-defined relations

Equation RELATION_CONSTRAINT_UP

$$ REL_{r,n,y} \leq relation\_upper_{r,n,y} $$

Equation RELATION_CONSTRAINT_LO

$$ REL_{r,n,y} \geq relation\_lower_{r,n,y} $$

Note: This page is generated from inline documentation in `MESSAGE/model_solve.gms`.

3.3.4 Solve statement workflow

This part of the code includes the perfect-foresight, myopic and rolling-horizon model solve statements including the required accounting of investment costs beyond the model horizon.

Perfect-foresight model

For the perfect foresight version of MESSAGE*ix*, include all years in the model horizon and solve the entire model. This is the standard option; the GAMS global variable `%foresight%=0` by default.

$$\min_{x} OBJ = \sum_{y \in Y} OBJ_y(x_y)$$

Recursive-dynamic and myopic model

For the myopic and rolling-horizon models, loop over horizons and iteratively solve the model, keeping the decision variables from prior periods fixed. This option is selected by setting the GAMS global variable `%foresight%` to a value greater than 0, where the value represents the number of years that the model instance is considering when iterating over the periods of the optimization horizon.

Loop over $\hat{y} \in Y$, solving

$$\min_{x} OBJ = \sum_{y \in \hat{Y}(\hat{y})} OBJ_y(x_y)$$

$$\text{s.t. } x_{y'} = x^*_{y'} \quad \forall\, y' < y$$

where $\hat{Y}(\hat{y}) = \{y \in Y | \, |\hat{y}| - |y| < optimization\_horizon\}$ and $x^*_{y'}$ is the optimal value of $x_{y'}$ in iteration $|y'|$ of the iterative loop.

The advantage of this implementation is that there is no need to 'store' the optimal values of all decision variables in additional reporting parameters - the last model solve automatically includes the results over the entire model horizon and can be imported via the ixmp interface.

Note: This page is generated from inline documentation in `MESSAGE/reporting.gms`.

3.3.5 Standard output reports

This part of the code contains the definitions and scripts for a number of standard output reports. These default reports will be created after every MESSAGE run.

Note: This page is generated from inline documentation in `MESSAGE/scaling_investment_costs.gms`.

3.3.6 Auxiliary investment parameters

Levelized capital costs

For the 'soft' relaxations of the dynamic constraints and the associated penalty factor in the objective function, we need to compute the parameter $levelized\_cost_{n,t,y}$.

$$
\begin{aligned}
levelized\_cost_{n,t,m,y,h} := {} & inv\_cost_{n,t,y} \cdot \frac{interestrate_y \cdot (1 + interestrate_y)^{|y|}}{(1 + interestrate_y)^{|y|} - 1} \\
& + fix\_cost_{n,t,y,y} \cdot \frac{1}{\sum_{h'} duration\_time_{h'} \cdot capacity\_factor_{n,t,y,y,h'}} \\
& + var\_cost_{n,t,y,y,m,h}
\end{aligned}
$$

where $|y| = technical\_lifetime_{n,t,y}$. This formulation implicitly assumes constant fixed and variable costs over time.

Warning: All soft relaxations of the dynamic activity constraint are disabled if the levelized costs are negative!

Construction time accounting

If the construction of new capacity takes a significant amount of time, investment costs have to be scaled up accordingly to account for the higher capital costs.

$$construction\_time\_factor_{n,t,y} = (1 + interestrate_y)^{|y|}$$

where $|y| = construction\_time_{n,t,y}$. If no construction time is specified, the default value of the investment cost scaling factor defaults to 1. The model assumes that the construction time only plays a role for the investment costs, i.e., each unit of new-built capacity is available instantaneously.

Comment: This formulation applies the discount rate of the vintage year (i.e., the year in which the new capacity becomes operational).

Investment costs beyond the model horizon

If the technical lifetime of a technology exceeds the model horizon Y^{model}, the model has to add a scaling factor to the investment costs ($end\_of\_horizon\_factor_{n,t,y}$). Assuming a constant stream of revenue (marginal value of the capacity constraint), this can be computed by annualizing investment costs from the condition that in an optimal solution, the investment costs must equal the discounted future revenues, if the investment variable $CAP\_NEW_{n,t,y} > 0$:

$$inv\_cost_{n,t,y^V} = \sum_{y \in Y^{lifetime}_{n,t,y^V}} df\_year_y \cdot \beta_{n,t},$$

Here, $\beta_{n,t} > 0$ is the dual variable to the capacity constraint (assumed constant over future periods) and $Y^{lifetime}_{n,t,y^V}$ is the set of periods in the lifetime of a plant built in period y^V. Then, the scaling factor $end\_of\_horizon\_factor_{n,t,y^V}$ can be derived as follows:

$$end\_of\_horizon\_factor_{n,t,y^V} := \frac{\sum_{y \in Y^{lifetime}_{n,t,y^V} \cap Y^{model}} df\_year_y}{\sum_{y' \in Y^{lifetime}_{n,t,y^V}} df\_year_{y'} + beyond\_horizon\_factor_{n,t,y^V}} \in (0, 1],$$

where the parameter $beyond\_horizon\_factor_{n,t,y^V}$ accounts for the discount factor beyond the overall model horizon (the set Y in contrast to the set $Y^{model} \subseteq Y$ of the periods included in the current model iteration (see the page on the recursive-dynamic model solution approach).

$$beyond\_horizon\_lifetime_{n,t,y^V} := \max\left\{0, economic\_lifetime_{n,t,y^V} - \sum_{y' \geq y^V} duration\_period_{y'}\right\}$$

$$beyond\_horizon\_factor_{n,t,y^V} := df\_year_{\widehat{y}} \cdot \frac{1}{(1 + interestrate_{\widehat{y}})^{|\widehat{y}|}} \cdot \frac{1 - \left(\frac{1}{1+interestrate_{\widehat{y}}}\right)^{|\widetilde{y}|}}{1 - \frac{1}{1+interestrate_{\widehat{y}}}}$$

where $\widehat{y}$ is the last period included in the overall model horizon, $|\widehat{y}| = period\_duration\_period_{\widehat{y}}$ and $|\widetilde{y}| = beyond\_horizon\_lifetime_{n,t,y^V}$.

If the interest rate is zero, i.e., $interestrate_{\widehat{y}} = 0$, the parameter $beyond\_horizon\_factor_{n,t,y^V}$ equals the remaining technical lifetime beyond the model horizon and the parameter $end\_of\_horizon\_factor_{n,t,y^V}$ equals the share of technical lifetime within the model horizon.

Possible extension: Instead of assuming $\beta_{n,t}$ to be constant over time, one could include a simple (linear) projection of $\beta_{n,t,y}$ beyond the end of the model horizon. However, this would likely require to include the computation of dual variables endogenously, and thus a mixed-complementarity formulation of the model.

Remaining installed capacity

The model has to take into account that the technical lifetime of a technology may not coincide with the cumulative period duration. Therefore, the model introduces the parameter $remaining\_capacity_{n,t,y^V,y}$ as a factor of remaining technical lifetime in the last period of operation divided by the duration of that period.

Note: This page is generated from inline documentation in `MACRO/macro_core.gms`.

3.3.7 MACRO - Mathematical formulation

MACRO is a macroeconomic model maximizing the intertemporal utility function of a single representative producer-consumer in each node (or macro-economic region). The optimization result is a sequence of optimal savings, investment, and consumption decisions. The main variables of the model are the capital stock, available labor, and commodity inputs, which together determine the total output of an economy according to a nested constant elasticity of substitution (CES) production function. End-use service demands in the (commercial) demand categories of MESSAGE is determined within the model, and is consistent with commodity supply curves, which are inputs to the model.

Notation declaration

The following short notation is used in the mathematical description relative to the GAMS code:

Math Notation	GAMS set & index notation	Description
n	node (or node_active in loops)	spatial node corresponding to the macro-economic MESSAGE regions
y	year	year (2005, 2010, 2020, ..., 2100)
s	sector	sector corresponding to the (commercial) end-use demands of MESSAGE

A listing of all parameters used in MACRO together with a decription can be found in the table below.

Parameter	Description
$duration\_period_y$	Number of years in time period y (forward diff)
$total\_cost_{n,y}$	Total system costs in region n and period y from MESSAGE model run
$enestart_{n,s,y}$	Consumption level of (commercial) end-use services s in region n and period y from MESSAGE model run
$eneprice_{n,s,y}$	Shadow prices of (commercial) end-use services s in region n and period y from MESSAGE model run
ϵ_n	Elasticity of substitution between capital-labor and total energy in region n
ρ_n	$\epsilon - 1/\epsilon$ where ϵ is the elasticity of subsitution in region n
$depr_n$	Annual depreciation rate in region n
α_n	Capital value share parameter in region n
a_n	Production function coefficient of capital and labor in region n
$b_{n,s}$	Production function coefficients of the different end-use sectors in region n, sector s and period y
$udf_{n,y}$	Utility discount factor in period year in region n and period y
$newlab_{n,y}$	New vintage of labor force in region n and period y
$grow_{n,y}$	Annual growth rates of potential GDP in region n and period y
$aeei_{n,s,y}$	Autonomous energy efficiency improvement (AEEI) in region n, sector s and period y
$fin\_time_{n,y}$	finite time horizon correction factor in utility function in region n and period y

Decision variables

Variable	Definition	Description
$K_{n,y}$	$K_{n,y} \geq 0 \; \forall n, y$	Capital stock in region n and period y
$KN_{n,y}$	$KN_{n,y} \geq 0 \; \forall n, y$	New Capital vintage in region n and period y
$Y_{n,y}$	$Y_{n,y} \geq 0 \; \forall n, y$	Total production in region n and period y
$YN_{n,y}$	$YN_{n,y} \geq 0 \; \forall n, y$	New production vintage in region n and period y
$C_{n,y}$	$C_{n,y} \geq 0 \; \forall n, y$	Consumption in region n and period y
$I_{n,y}$	$I_{n,y} \geq 0 \; \forall n, y$	Investment in region n and period y
$PHYSENE_{n,s,y}$	$PHYSENE_{n,s,y} \geq 0 \; \forall n, s, y$	Physical end-use service use in region n, sector s and period y
$PRODENE_{n,s,y}$	$PRODENE_{n,s,y} \geq 0 \; \forall n, s, y$	Value of end-use service in the production function in region n, sector s and period y
$NEWENE_{n,s,y}$	$NEWENE_{n,s,y} \geq 0 \; \forall n, s, y$	New end-use service in the production function in region n, sector s and period y
$EC_{n,y}$	$EC \in [-\infty..\infty]$	Approximation of system costs based on MESSAGE results
$UTILITY$	$UTILITY \in [-\infty..\infty]$	Utility function (discounted log of consumption)

Equation UTILITY_FUNCTION

The utility function which is maximized sums up the discounted logarithm of consumption of a single representative producer-consumer over the entire time horizon of the model.

$$
UTILITY = \sum_n \left(\sum_{y|((ord(y)>1) \wedge (ord(y)<|y|))} udf_{n,y} \cdot \log(C_{n,y}) \cdot duration\_period_y \right.
$$

$$
\left. + \sum_{y|(ord(y)=|y|)} udf_{n,y} \cdot \log(C_{n,y}) \cdot \left(duration\_period_{y-1} + \frac{1}{FIN\_TIME_{n,y}} \right) \right)
$$

The utility discount rate for period y is set to $drate_n - grow_{n,y}$, where $drate_n$ is the discount rate used in MESSAGE, typically set to 5%, and $grow$ is the potential GDP growth rate. This choice ensures that in the steady state, the optimal growth rate is identical to the potential GDP growth rates $grow$. The values for the utility discount rates are chosen for descriptive rather than normative reasons. The term $\frac{duration\_period_y + duration\_period_{y-1}}{2}$ mutliples the discounted logarithm of consumption with the period length. The final period is treated separately to include a correction factor $\frac{1}{FIN\_TIME_{n,y}}$ reflecting the finite time horizon of the model. Note that the sum over nodes $node\_active$ is artificial, because $node\_active$ only contains one element.

Equation CAPITAL_CONSTRAINT

The following equation specifies the allocation of total production among current consumption $C_{n,y}$, investment into building up capital stock excluding the sectors represented in MESSAGE $I_{n,y}$ and the MESSAGE system costs $EC_{n,y}$ which are derived from a previous MESSAGE model run. As described in [4], the first-order optimality conditions lead to the Ramsey rule for the optimal allocation of savings, investment and consumption over time.

$$Y_{n,y} = C_{n,y} + I_{r,y} + EC_{n,y} \qquad \forall n, y$$

Equation NEW_CAPITAL

The accumulation of capital in the sectors not represented in MESSAGE is governed by new capital stock equation. Net capital formation $KN_{n,y}$ is derived from gross investments $I_{n,y}$ minus depreciation of previsouly existing capital stock.

$$KN_{n,y} = duration\_period_y \cdot I_{n,y} \qquad \forall n, y > 1$$

Here, the initial boundary condition for the base year y_0 implies for the investments that $I_{n,y_0} = (grow_{n,y_0} + depr_n) \cdot kgdp_n \cdot gdp_{n,y_0}$.

Equation NEW_PRODUCTION

MACRO employs a nested constant elasticity of substitution (CES) production function with capital, labor and the (commercial) end-use services represented in MESSAGE as inputs. This version of the production function is equaivalent to that in MARKAL-MACRO.

$$YN_{n,y} = \left(a_n \cdot KN_{n,y}^{(\rho_n \cdot \alpha_n)} \cdot newlab_{n,y}^{(\rho_n \cdot (1-\alpha_n))} + \sum_s (b_{n,s} \cdot NEWENE_{n,s,y}^{\rho_n}) \right)^{\frac{1}{\rho_n}} \qquad \forall n, y > 1$$

Equation TOTAL_CAPITAL

Equivalent to the total production equation above, the total capital stock, again excluding those sectors which are modeled in MESSAGE, is then simply a summation of capital stock in the previous period $y - 1$, depreciated with the depreciation rate $depr_n$, and the capital stock added in the current period y.

$$K_{n,y} = K_{n,y-1} \cdot (1 - depr_n)^{duration\_period_y} + KN_{n,y} \qquad \forall n, y > 1$$

Equation TOTAL_PRODUCTION

Total production in the economy (excluding energy sectors) is the sum of production from assets that were already existing in the previous period $y - 1$, depreciated with the depreciation rate $depr_n$, and the new vintage of production

from period y.

$$Y_{n,y} = Y_{n,y-1} \cdot (1 - depr_n)^{duration\_period_y} + YN_{n,y} \qquad \forall n, y > 1$$

Equation NEW_ENERGY

Total energy production (across the six commerical energy demands s) is the sum of production from all assets that were already existing in the previous period $y - 1$, depreciated with the depreciation rate $depr_n$, and the the new vintage of energy production from period y.

$$PRODENE_{n,s,y} = PRODENE_{n,s,y-1} \cdot (1 - depr_n)^{duration\_period_y} + NEWENE_{n,s,y} \qquad \forall n, s, y > 1$$

Equation ENERGY_SUPPLY

The relationship below establishes the link between physical energy $PHYSENE_{r,s,y}$ as accounted in MESSAGE for the six commerical energy demands s and energy in terms of monetary value $PRODENE_{n,s,y}$ as specified in the production function of MACRO.

$$PHYSENE_{n,s,y} \geq PRODENE_{n,s,y} \cdot aeei\_factor_{n,s,y} \qquad \forall n, s, y > 1$$

The cumulative effect of autonomous energy efficiency improvements (AEEI) is captured in $aeei\_factor_{n,s,y} = aeei\_factor_{n,s,y-1} \cdot (1 - aeei_{n,s,y})_y^{duration\_period}$ with $aeei\_factor_{n,s,y=1} = 1$. Therefore, choosing the $aeei_{n,s,y}$ coefficients appropriately offers the possibility to calibrate MACRO to a certain energy demand trajectory from MESSAGE.

Equation COST_ENERGY

Energy system costs are based on a previous MESSAGE model run. The approximation of energy system costs in vicinity of the MESSAGE solution are approximated by a Taylor expansion with the first order term using shadow prices $eneprice_{s,y,n}$ of the MESSAGE model's solution and a quadratic second-order term.

$$\begin{aligned} EC_{n,y} = & total\_cost_{n,r} \\ & + \sum_s eneprice_{s,y,n} \cdot (PHYSENE_{n,s,y} - enestart_{s,y,n}) \\ & + \sum_s \frac{eneprice_{s,y,n}}{enestart_{s,y,n}} \cdot (PHYSENE_{n,s,y} - enestart_{s,y,n})^2 \qquad \forall n, y > 1 \end{aligned}$$

Equation TERMINAL_CONDITION

Given the finite time horizon of MACRO, a terminal constraint needs to be applied to ensure that investments are chosen at an appropriate level, i.e. to replace depriciated capital and provide net growth of capital stock beyond MACRO's time horizon [4]. The goal is to avoid to the extend possible model artifacts resulting from this finite time horizon cutoff.

$$K_{n,y} \cdot (grow_{n,y} + depr_n) \leq I_{n,y} \qquad \forall n, y = lastyear$$

3.4 Developing MESSAGE*ix* models

Developing a valid, scientific MESSAGE*ix* model requires careful use of the framework features. This section provides guidelines for how to make some common model design choices.

3.4.1 Efficiency - output- vs. input defined technologies

There is no obvious approach whether a model should be formulated in a way that treats technologies as parametrized to input or output commodities/fuels - power plant parameters are usually understood as output-based (per unit of electricity generated), while refinery parameters are usually based on input fuels (per unit of input commodity processed. Things become even trickier when technologies have multiple inputs or outputs. Standardizing the methodology and assumptions can become quite a challenge.

For the implementation of MESSAGE*ix*, we opted to formulate the model in an agnostic manner, so that for each technology, the most "appropriate" parametrization can be applied. As an additional benefit, we do not need to define an explicit efficiency parameter or "main" input and output fuels.

The recommended approach is illustrated below for multiple examples. The decision variables CAP_NEW, CAP and ACT as well as all bounds are always understood to be in the same units. All cost parameters also have to be provided in monetary units per these units - there is no "automatic rescaling" done either within the ixmp API or in the GAMS implementation pre- or postprocessing.

Example 1 - Power plants

Technical specifications of power plants are commonly stated in terms of electricity generated (output). Therefore, the decision variables should be understood as outputs, with the parameter $output = 1$ and parameter $input = \frac{1}{efficiency}$. This may seem counter-intuitive at first, but the clear advantage is that all technical parameters can be immediately related to values found in the literature.

Example 2 - Refineries

For crude oil refineries, it is more common to scale costs and emissions in terms of crude oil input quantities. Hence, the parameter $input = 1$ and the output parameters (usually for multiple different oil products) should be set accordingly.

The decision variables and bounds are then implicitly understood as input-based.

An alternative would be to parametrize a refinery based on outputs, but considering that there are multiple outputs (in fixed proportions), the sum of output parameters over all products should be set to 1, i.e., $\sum_c output_c = 1$. The input of crude oil should then include the losses during the refining process, $input > 1$.

Example 3 - Combined power- and heat plants

As a third option, technical specifications of combined heat- and power plants are usually also given with regard to electricity generated under the assumption that the electricity generated is maximized. Then, as in example 1, the capacity and activity variables should be understood as electricity generated.

Assuming that such a plant usually has (at least) two modes of operation, these modes could be parametrized as follows:

$input = \frac{1}{efficiency}$

$output_{'M1','electricity'} = 1$ and $output_{'M1','heat'} = 0.2$

$output_{'M2','electricity'} = 0.5$ and $output_{'M2','heat'} = 3$.

Note that the activity level in mode 'M2' has an odd interpretation - the amount of electricity generated if electricity generation were maximized. The sum of outputs is greater than 1 in either mode. However, we believe that this approach at least has the benefit of a parametrization that can be directly related to technical reports.

3.4.2 Debugging and data validation

Finding the cause for infeasibilities or counter-intuitive results in large-scale numerical models is not trivial. For this reason, the MESSAGE*ix* framework includes a number of features to simplify debugging and pre-processing data validation.

Pre-processing data validation

The data validation checks are included in the file `model/MESSAGE/data_load.gms`. If the data validation fails, an error message is written to the log file.

Identification of infeasibilities

The MESSAGE*ix* framework includes the option to "relax" the most common constraints, simultaneously adding a penalty term for the relaxation to the objective function. Solving the relaxed version of the model can help to identify incompatible constraints or input data errors causing infeasible models.

The relaxations can be activated by blocks/types of equations by setting the respective global variables (`$SETGLOBAL` in GAMS) in `MESSAGE_master.gms` or by calling `MESSAGE_run.gms` passing the global variables as command-line arguments.

3.4.3 Postprocessing and reporting

> **Warning:** `message_ix.reporting` is **experimental** in message_ix 1.2 and only supports Python 3. The API and functionality may change without advance notice or a deprecation period in subsequent releases.

The *ix modeling platform* provides powerful features to perform calculations and other postprocessing after a *message_ix.Scenario* has been solved by the associated model. The MESSAGE*ix* framework uses these features to provide zero-configuration reporting of models built on the framework.

These features are accessible through `Reporter`, which can produce multiple **reports** from one or more Scenarios. A report is identified by a **key** (usually a string), and may...

- perform arbitrarily complex calculations while intelligently handling units;
- read and make use of data that is 'exogenous' to (not included in) a Scenario;
- produce output as Python or R objects (in code), or to files or databases;
- calculate only a requested subset of quantities; and
- much, much more!

Terminology

`ixmp.reporting` handles numerical **quantities**, which are scalar (0-dimensional) or array (1 or more dimensions) data with optional associated units. *ixmp* parameters, scalars, equations, and time-series data all become quantities for the purpose of reporting.

Every quantity and report is identified by a **key**, which is a `str` or other hashable object. Special keys are used for multidimensional quantities. For instance: the MESSAGE*ix* parameter `resource_cost`, defined with the dimensions (node *n*, commodity *c*, grade *g*, year *y*) is identified by the key `'resource_cost:n-c-g-y'`. When summed across the grade/*g* dimension, it has dimensions *n*, *c*, *y* and is identified by the key `'resource_cost:n-c-y'`.

Non-model[1] quantities and reports are produced by **computations**, which are atomic tasks that build on other computations. The most basic computations—for instance, `resource_cost:n-c-g-y`—simply retrieve raw/unprocessed data from a *`message_ix.Scenario`* and return it as a *`Quantity`*. Advanced computations can depend on many quantities, and/or combine quantities together into a structure like a document or spreadsheet. Computations are defined in *`ixmp.reporting.computations`* and `message_ix.reporting.computations`, but most common computations can be added using the methods of `Reporter`.

Basic usage

A basic reporting workflow has the following steps:

1. Obtain a *`message_ix.Scenario`* object from an *`ixmp.Platform`*.

2. Use `from_scenario()` to create a Reporter object.

3. (optionally) Use `Reporter` built-in methods or advanced features to add computations to the reporter.

4. Use *`get()`* to retrieve the results (or trigger the effects) of one or more computations.

```
>>> from ixmp import Platform
>>> from message_ix import Scenario, Reporter
>>>
>>> mp = Platform()
>>> scen = Scenario(scen)
>>> rep = Reporter.from_scenario(scen)
>>> rep.get('all')
```

Note: `Reporter` stores defined computations, but these **are not executed** until *`get()`* is called—or the results of one computation are required by another. This allows the Reporter to skip unneeded (and potentially slow) computations. A Reporter may contain computations for thousands of model quantities and derived quantities, but a call to *`get()`* may only execute a few of these.

Customization

A Reporter prepared with `from_scenario()` always contains a key `scenario`, referring to the Scenario to be reported.

The method *`Reporter.add()`* can be used to add *arbitrary* Python code that operates directly on the Scenario object:

```
>>> def my_custom_report(scenario):
>>>     """Function with custom code that manipulates the *scenario*."""
>>>     print('foo')
>>>
>>> rep.add('custom', (my_custom_report, 'scenario'))
>>> rep.get('custom')
foo
```

In this example, the function `my_custom_report()` *could* run to thousands of lines; read to and write from multiple files; invoke other programs or Python scripts; etc.

In order to take advantage of the performance-optimizing features of the Reporter, however, such calculations can be instead composed from atomic (i.e. small, indivisible) computations.

---

[1] i.e. quantities that do not exist within the mathematical formulation of the model itself, and do not affect its solution.

Reporters

reporting.**configure**(***config*)
> Configure reporting globally.

> Valid configuration keys include:

> - *units*:

>> - *define*: a string, passed to `pint.UnitRegistry.define()`.

>> - *replace*: a mapping from str to str, used to replace units before they are parsed by pints

> **Warns   UserWarning** – If *config* contains unrecognized keys.

class ixmp.reporting.**Reporter**(***kwargs*)
> Class for generating reports on `ixmp.Scenario` objects.

> A Reporter is used to postprocess data from from one or more `ixmp.Scenario` objects. The `get()` method can be used to:

> - Retrieve individual **quantities**. A quantity has zero or more dimensions and optional units. Quantities include the 'parameters', 'variables', 'equations', and 'scalars' available in an `ixmp.Scenario`.

> - Generate an entire **report** composed of multiple quantities. A report may:

>> - Read in non-model or exogenous data,

>> - Trigger output to files(s) or a database, or

>> - Execute user-defined methods.

> Every report and quantity (including the results of intermediate steps) is identified by a `utils.Key`; all the keys in a Reporter can be listed with `keys()`.

> Reporter uses a graph data structure to keep track of **computations**, the atomic steps in postprocessing: for example, a single calculation that multiplies two quantities to create a third. The graph allows `get()` to perform *only* the requested computations. Advanced users may manipulate the graph directly; but common reporting tasks can be handled by using Reporter methods:

add(key, computation[, strict])	Add *computation* to the Reporter under *key*.
add_file(path[, key])	Add exogenous quantities from *path*.
aggregate(qty, tag, dims_or_groups[, . . .])	Add a computation that aggregates *qty*.
apply(generator, *keys)	Add computations from *generator* applied to *key*.
configure([path])	Configure the Reporter.
describe([key])	Return a string describing the computations that produce *key*.
disaggregate(qty, new_dim[, method, args])	Add a computation that disaggregates *var* using *method*.
finalize(scenario)	Prepare the Reporter to act on *scenario*.
full_key(name)	Return the full-dimensionality key for *name*.
get([key])	Execute and return the result of the computation *key*.
read_config(path)	Configure the Reporter with information from a YAML file at *path*.
visualize(filename, **kwargs)	Generate an image describing the reporting structure.
write(key, path)	Write the report *key* to the file *path*.

> **graph = {'filters': None}**
>> A dask-format graph.

add(*key*, *computation*, *strict=False*)

Add *computation* to the Reporter under *key*.

>**Parameters**
>
>>- **key** (`hashable`) – A string, Key, or other value identifying the output of *task*.
>>
>>- **computation** (`object`) – One of:
>>
>>>1. any existing key in the Reporter.
>>>
>>>2. any other literal value or constant.
>>>
>>>3. a task, i.e. a tuple with a callable followed by one or more computations.
>>>
>>>4. A list containing one or more of #1, #2, and/or #3.
>>
>>- **strict** (`bool, optional`) – If True, *key* must not already exist in the Reporter, and any keys referred to by *computation* must exist.
>
>**Raises** `KeyError` – If *key* is already in the Reporter, or any key referred to by *computation* does not exist.

add() may be used to:

- Provide an alias from one *key* to another:

```
>>> r.add('aliased name', 'original name')
```

- Define an arbitrarily complex computation in a Python function that operates directly on the `ixmp.Scenario`:

```
>>> def my_report(scenario):
>>>     # many lines of code
>>>     return 'foo'
>>> r.add('my report', (my_report, 'scenario'))
>>> r.finalize(scenario)
>>> r.get('my report')
foo
```

Note: Use care when adding literal `str` values (2); these may conflict with keys that identify the results of other computations.

---

add_file(*path*, *key=None*)

Add exogenous quantities from *path*.

A file at a path like '/path/to/foo.ext' is added at the key `'file:foo.ext'`.

See also:

`ixmp.reporting.computations.load_file()`

add_product(*name*, **quantities*, *sums=True*)

Add a computation that takes the product of *quantities*.

>**Parameters**
>
>>- **name** (`str`) – Name of the new quantity.
>>
>>- **sums** (`bool, optional`) – If `True`, all partial sums of the new quantity are also added.
>
>**Returns** The full key of the new quantity.

> **Return type** *Key*

aggregate(*qty*, *tag*, *dims_or_groups*, *weights=None*, *keep=True*)

Add a computation that aggregates *qty*.

> **Parameters**
>
> - **qty** (`Key` or str) – Key of the quantity to be disaggregated.
>
> - **tag** (`str`) – Additional string to add to the end the key for the aggregated quantity.
>
> - **dims_or_groups** (`str or iterable of str or dict`) – Name(s) of the dimension(s) to sum over, or nested dict.
>
> - **weights** (`xr.DataArray`) – Weights for weighted aggregation.
>
> **Returns** The key of the newly-added node.
>
> **Return type** `Key`

apply(*generator*, **keys*)

Add computations from *generator* applied to *key*.

> **Parameters**
>
> - **generator** (`callable`) – Function to apply to *keys*. Must yield a sequence (0 or more) of (*key*, *computation*), which are added to the *graph*.
>
> - **keys** (`hashable`) – The starting key(s)

check_keys(**keys*)

Check that *keys* are in the Reporter.

If any of *keys* is not in the Reporter, KeyError is raised. Otherwise, a list is returned with either the key from *keys*, or the corresponding *full_key()*.

configure(*path=None*, ***config*)

Configure the Reporter.

Accepts a *path* to a configuration file and/or keyword arguments. Configuration keys loaded from file are replaced by keyword arguments.

Valid configuration keys include:

- *default*: the default reporting key; sets *default_key*.

- *filters*: a `dict`, passed to *set_filters()*.

- *files*: a `dict` mapping keys to file paths.

- *alias*: a `dict` mapping aliases to original keys.

> **Warns** **UserWarning** – If *config* contains unrecognized keys.

default_key = None

The default reporting key.

describe(*key=None*)

Return a string describing the computations that produce *key*.

If *key* is not provided, all keys in the Reporter are described.

disaggregate(*qty*, *new_dim*, *method='shares'*, *args=[]*)

Add a computation that disaggregates *var* using *method*.

> **Parameters**

- **var** (`hashable`) – Key of the variable to be disaggregated.

- **new_dim** (`str`) – Name of the new dimension of the disaggregated variable.

- **method** (`callable or str`) – Disaggregation method. If a callable, then it is applied to *var* with any extra *args*. If then a method named 'disaggregate_{method}' is used.

- **args** (`list, optional`) – Additional arguments to the *method*. The first element should be the key for a quantity giving shares for disaggregation.

> **Returns** The key of the newly-added node.

> **Return type** *Key*

finalize(*scenario*)

Prepare the Reporter to act on *scenario*.

The *Scenario* object *scenario* is associated with the key `'scenario'`. All subsequent processing will act on data from this *scenario*.

classmethod from_scenario(*scenario*, ***kwargs*)

Create a Reporter by introspecting *scenario*.

> **Parameters**
>
> - **scenario** (`ixmp.Scenario`) – Scenario to introspect in creating the Reporter.
>
> - **kwargs** (`optional`) – Passed to `Scenario.configure()`.
>
> **Returns**
>
> A Reporter instance containing:
>
> - A 'scenario' key referring to the *scenario* object.
>
> - Each parameter, equation, and variable in the *scenario*.
>
> - All possible aggregations across different sets of dimensions.
>
> - Each set in the *scenario*.
>
> **Return type** *Reporter*

full_key(*name*)

Return the full-dimensionality key for *name*.

An ixmp variable 'foo' indexed by a, c, n, q, and x is available in the Reporter at `'foo:a-c-n-q-x'`. `full_key('foo')` retrieves this *Key*.

get(*key=None*)

Execute and return the result of the computation *key*.

Only *key* and its dependencies are computed.

> **Parameters key** (`str, optional`) – If not provided, *default_key* is used.
>
> **Raises** `ValueError` – If *key* and *default_key* are both `None`.

read_config(*path*)

Configure the Reporter with information from a YAML file at *path*.

See *configure()*.

set_filters(***filters*)

Apply *filters* ex ante (before computations occur).

filters has the same form as the argument of the same name to *ixmp.Scenario.par()* and analogous methods. A value of `None` will clear the filter for the named dimension.

visualize(*filename*, ***kwargs*)

>   Generate an image describing the reporting structure.

>   This is a shorthand for `dask.visualize()`. Requires [graphviz](#).

write(*key*, *path*)

>   Write the report *key* to the file *path*.

Computations

Computations from ixmp

Elementary computations for reporting.

[aggregate](#)(quantity, groups, keep)	Aggregate *quantity* by *groups*.
[disaggregate_shares](#)(quantity, shares)	Disaggregate *quantity* by *shares*.
[load_file](#)(path)	Read the file at *path* and return its contents.
[make_dataframe](#)(*quantities)	Concatenate *quantities* into a single pd.DataFrame.
[write_report](#)(quantity, path)	Write the report identified by *key* to the file at *path*.

`ixmp.reporting.computations.`**aggregate**(*quantity*, *groups*, *keep*)

>   Aggregate *quantity* by *groups*.

`ixmp.reporting.computations.`**disaggregate_shares**(*quantity*, *shares*)

>   Disaggregate *quantity* by *shares*.

`ixmp.reporting.computations.`**make_dataframe**(**quantities*)

>   Concatenate *quantities* into a single pd.DataFrame.

`ixmp.reporting.computations.`**load_file**(*path*)

>   Read the file at *path* and return its contents.

>   Some file formats are automatically converted into objects for direct use in reporting code:

>   - *csv*: converted to `xarray.DataArray`. CSV files must have a 'value' column; all others are treated as indices.

`ixmp.reporting.computations.`**sum**(*quantity*, *weights=None*, *dimensions=None*)

>   Sum *quantity* over *dimensions*, with optional *weights*.

`ixmp.reporting.computations.`**write_report**(*quantity*, *path*)

>   Write the report identified by *key* to the file at *path*.

Utilities

class `ixmp.reporting.utils.`**Key**(*name*, *dims=[]*, *tag=None*)

>   A hashable key for a quantity that includes its dimensionality.

>   Quantities in a `Scenario` can be indexed by one or more dimensions. For example, a parameter with three dimensions can be initialized with:

```
>>> scenario.init_par('foo', ['a', 'b', 'c'], ['apple', 'bird', 'car'])
```

>   Computations for this scenario might use the quantity `foo` in different ways:

>   1. in its full resolution, i.e. indexed by a, b, and c;

>   2. aggregated (e.g. summed) over any one dimension, e.g. aggregated over c and thus indexed by a and b;

3. aggregated over any two dimensions; etc.

A Key for (1) will hash, display, and evaluate as equal to `'foo:a-b-c'`. A Key for (2) corresponds to `'foo:a-b'`, and so forth.

Keys may be generated concisely by defining a convenience method:

```
>>> def foo(dims):
>>>     return Key('foo', dims.split(''))
>>> foo('a b')
foo:a-b
```

classmethod from_str_or_key(*value*, *drop=None*, *append=None*, *tag=None*)
    Return a new Key from *value*.

iter_sums()
    Yield (key, task) for all possible partial sums of the Key.

classmethod product(*new_name*, **keys*)
    Return a new Key that has the union of dimensions on *keys*.

    Dimensions are ordered by their first appearance.

class `ixmp.reporting.utils.`**AttrSeries**(**args*, ***kwargs*)
    `pandas.Series` subclass imitating `xarray.DataArray`.

    Future versions of `ixmp.reporting` will use `xarray.DataArray` as *Quantity*; however, because `xarray` currently lacks sparse matrix support, ixmp quantities may be too large for memory.

    The AttrSeries class provides similar methods and behaviour to `xarray.DataArray`, such as an *attrs* dictionary for metadata, so that *ixmp.reporting.computations* methods can use xarray-like syntax.

`ixmp.reporting.utils.`**Quantity**
    alias of *ixmp.reporting.utils.AttrSeries*

`ixmp.reporting.utils.`**clean_units**(*input_string*)
    Tolerate messy strings for units.

    Handles two specific cases found in MESSAGE*ix* test cases:

    - Dimensions enclosed in '[]' have these characters stripped.

    - The '%' symbol cannot be supported by pint, because it is a Python operator; it is translated to 'percent'.

`ixmp.reporting.utils.`**collect_units**(**args*)
    Return an list of '_unit' attributes for *args*.

`ixmp.reporting.utils.`**data_for_quantity**(*ix_type*, *name*, *column*, *scenario*, *filters=None*)
    Retrieve data from *scenario*.

    Parameters

    - **ix_type**(`'equ' or 'par' or 'var'`) – Type of the ixmp object.

    - **name** (`str`) – Name of the ixmp object.

    - **column**(`'mrg' or 'lvl' or 'value'`) – Data to retrieve. 'mrg' and 'lvl' are valid only for `ix_type='equ'`, and 'level' otherwise.

    - **scenario** (`ixmp.Scenario`) – Scenario containing data to be retrieved.

    - **filters**(`dict, optional`) – Mapping from dimensions to iterables of allowed values along each dimension.

    Returns Data for *name*.

> **Return type** [*Quantity*](#)

ixmp.reporting.utils.**keys_for_quantity**(*ix_type*, *name*, *scenario*)
> Iterate over keys for *name* in *scenario*.

3.4.4 Model-building tools

Add model years to an existing Scenario

Description

This tool adds new modeling years to an existing [*ixmp.Scenario*](#) (hereafter "reference scenario"). For instance, in a scenario define with:

```
history = [690]
model_horizon = [700, 710, 720]
sc_ref.add_horizon({'year': history + model_horizon,
                    'firstmodelyear': model_horizon[0]})
```

...additional years can be added:

```
sc_new = message_ix.Scenario(mp, sc_ref.model, sc_ref.scenario,
                             version='new')
add_year(sc_ref, sc_new, [705, 712, 718, 725])
```

At this point, `sc_new` will have the years [700, 705, 710, 712, 718, 720, 725], and original or interpolated data for all these years in all parameters.

The tool operates by creating a new empty Scenario (hereafter "new scenario") and:

- Copying all **sets** from the reference scenario, adding new time steps to relevant sets (e.g., adding 2025 between 2020 and 2030 in the set `year`)

- Copying all **parameters** from the reference scenario, adding new years to relevant parameters, and calculating missing values for the added years.

Features

- It can be used for any MESSAGE scenario, from tutorials, country-level, and global models.

- The new years can be consecutive, between existing years, and/or after the model horizon.

- The user can define for what regions and parameters the new years should be added. This saves time when adding the new years to only one parameter of the reference scenario, when other parameters have previously been successfully added to the new scenario.

Usage

The tool can be used either:

1. Directly from the command line:

```
$ python -m message_ix.tools.add_year \
    --model_ref MESSAGE_Model \
    --scen_ref baseline \
    --years_new 2015,2025,2035,2045
```

For the full list of input arguments, run:

```
$ python -m message_ix.tools.add_year --help
```

2. By calling the function *message_ix.tools.add_year.add_year()* from a Python script.

Technical details

1. An existing scenario is loaded and the desired new years are specified.

2. A new (empty) scenario is created for adding the new years.

3. The new years are added to the relevant sets, `year` and `type_year`.

    • The sets `firstmodelyear`, `lastmodelyear`, `baseyear_macro`, and `initializeyear_macro` are modified, if needed.

    • The set `cat_year` is modified for the new years.

4. The new years are added to the index sets of relevant parameters, and the missing data for the new years are calculated based on interpolation of adjacent data points. The following steps are applied:

    a. Each non-empty parameter is loaded from the reference scenario.

    b. The year-related indexes (0, 1, or 2) of the parameter are identified.

    c. The new years are added to the parameter, and the missing data is calculated based on the number of year-related indexes. For example:

        • The parameter `inv_cost` has index `year_vtg`, to which the new years are added.

        • The parameter `output` has indices `year_act` and `year_vtg`. The new years are added to *both* of these dimensions.

    d. Missing data is calculated by interpolation.

    e. For parameters with 2 year-related indices (e.g. `output`), a final check is applied so ensure that the vintaging is correct. This step is done based on the lifetime of each technology.

5. The changes are committed and saved to the new scenario.

> **Warning:** The tool does not ensure that the new scenario will solve after adding the new years. The user needs to load the new scenario, check some key parameters (like bounds) and solve the new scenario.

API reference

Add model years to an existing Scenario.

message_ix.tools.add_year.**add_year**(*sc_ref*, *sc_new*, *years_new*, *firstyear_new=None*, *lastyear_new=None*, *macro=False*, *baseyear_macro=None*, *parameter='all'*, *region='all'*, *rewrite=True*, *unit_check=True*, *extrapol_neg=None*, *bound_extend=True*)

Add years to *sc_ref* to produce *sc_new*.

add_year() does the following:

1. calls *add_year_set()* to add and modify required sets.

2. calls *add_year_par()* to add new years and modifications to each parameter if needed.

> **Parameters**
>
> - **sc_ref** (`ixmp.Scenario`) – Reference scenario.
> - **sc_new** (`ixmp.Scenario`) – New scenario.
> - **yrs_new** (`list of int`) – New years to be added.
> - **firstyear_new** (`int, optional`) – New first model year for new scenario.
> - **macro** (`bool`) – Add new years to parameters of the MACRO model.
> - **baseyear_macro** (`int`) – New base year for the MACRO model.
> - **parameter** (`list of str or 'all'`) – Parameters for adding new years.
> - **rewrite** (`bool`) – Permit rewriting a parameter in new scenario when adding new years.
> - **check_unit** (`bool`) – Harmonize the units for each commodity, if there is inconsistency across model years.
> - **extrapol_neg** (`float`) – When extrapolation produces negative values, replace with a multiple of the value for the previous timestep.
> - **bound_extend** (`bool`) – Duplicate data from the previous timestep when there is only one data point for interpolation (e.g., permitting the extension of a bound to 2025, when there is only one value in 2020).

message_ix.tools.add_year.**add_year_par**(*sc_ref*, *sc_new*, *yrs_new*, *parname*, *region_list*, *firstyear_new*, *extrapolate=False*, *rewrite=True*, *unit_check=True*, *extrapol_neg=None*, *bound_extend=True*)

Add new years to parameters.

This function adds additional years to a parameter. The value of the parameter for additional years is calculated mainly by interpolating and extrapolating data from existing years.

See *add_year()* for parameter descriptions.

message_ix.tools.add_year.**add_year_set**(*sc_ref*, *sc_new*, *years_new*, *firstyear_new=None*, *lastyear_new=None*, *baseyear_macro=None*)

Add new years to sets.

add_year_set() adds additional years to an existing scenario, by starting to make a new scenario from scratch. After modification of the year-related sets, all other sets are copied from *sc_ref* to *sc_new*.

See *add_year()* for parameter descriptions.

message_ix.tools.add_year.**interpolate_1d**(*df*, *yrs_new*, *horizon*, *year_col*, *value_col='value'*, *extrapolate=False*, *extrapol_neg=None*, *bound_extend=True*)

Interpolate data with one year dimension.

This function receives a parameter data as a dataframe, and adds new data for the additonal years by interpolation and extrapolation.

> **Parameters**
>
> - **df** (*pandas.DataFrame*) – The dataframe of the parameter to which new years to be added.
>
> - **yrs_new** (*list of int*) – New years to be added.
>
> - **horizon** (*list of int*) – The horizon of the reference scenario.
>
> - **year_col** (*str*) – The header of the column to which the new years should be added, e.g. *'year_act'*.
>
> - **value_col** (*str*) – The header of the column containing values.
>
> - **extrapolate** (*bool*) – Allow extrapolation when a new year is outside the parameter years.
>
> - **extrapol_neg** (*bool*) – Allow negative values obtained by extrapolation.
>
> - **bound_extend** (*bool*) – Allow extrapolation of bounds for new years

message_ix.tools.add_year.**interpolate_2d**(*df*, *yrs_new*, *horizon*, *year_ref*, *year_col*, *tec_list*, *par_tec*, *value_col='value'*, *extrapolate=False*, *extrapol_neg=None*, *year_diff=None*, *bound_extend=True*)

Interpolate parameters with two dimensions related year.

This function receives a dataframe that has 2 time-related columns (e.g., "input" or "relation_activity"), and adds new data for the additonal years in both time-related columns by interpolation and extrapolation.

> **Parameters**
>
> - **df** (*pandas.DataFrame*) – The dataframe of the parameter to which new years to be added.
>
> - **yrs_new** (*list of int*) – New years to be added.
>
> - **horizon** (*list of int*) – The horizon of the reference scenario.
>
> - **year_ref** (*str*) – The header of the first column to which the new years should be added, e.g. *'year_vtg'*.
>
> - **year_col** (*str*) – The header of the column to which the new years should be added, e.g. *'year_act'*.
>
> - **tec_list** (*list of str*) – List of technologies in the parameter `technical_lifetime`.
>
> - **par_tec** (*pandas.DataFrame*) – Parameter `technical_lifetime`.
>
> - **value_col** (*str*) – The header of the column containing values.
>
> - **extrapolate** (*bool*) – Allow extrapolation when a new year is outside the parameter years.
>
> - **extrapol_neg** (*bool*) – Allow negative values obtained by extrapolation.
>
> - **year_diff** (*list of int*) – List of model years with different time intervals before and after them
>
> - **bound_extend** (*bool*) – Allow extrapolation of bounds for new years

message_ix.tools.add_year.**intpol**(*y1*, *y2*, *x1*, *x2*, *x*)

Interpolate between (*x1*, *y1*) and (*x2*, *y2*) at *x*.

Parameters

- **y2** (*y1,*) –

- **x2, x** (*x1,*) –

message_ix.tools.add_year.**mask_df**(*df*, *index*, *count*, *value*)
> Create a mask for removing extra values from *df*.

message_ix.tools.add_year.**slice_df**(*df*, *idx*, *level*, *locator*, *value*)
> Slice a MultiIndex DataFrame and set a value to a specific level.

> Parameters

- **df** (*pd.DataFrame*) –

- **idx** (*list of indices*) –

- **level** (*str*) –

- **locator** (*list*) –

- **value** (*int or str*) –

message_ix.tools.add_year.**unit_uniform**(*df*)
> Make units in *df* uniform.

Using and contributing to MESSAGE*ix*

MESSAGE*ix* and the *ix modeling platform* are licensed under the APACHE 2.0 open-source license.

Anyone is encouraged to use the framework to develop energy system and integrated assessment models! Please see the *User guidelines and notice* for using the framework in scientific research. Contributions to the framework itself, which enable new features across all models, are also welcome.

4.1 User guidelines and notice

We ask that you take the following four actions whenever you:

- **use** the MESSAGE*ix* framework, *ix modeling platform*, or any model(s) you have built using these tools

- to **produce** any scientific publication, technical report, website, or other **publicly-available material**.

The aim of this request is to ensure good scientific practice and collaborative development of the platform.

4.1.1 1. Understand the code license

Use the most recent version of MESSAGE*ix* from the Github repository. Specify clearly which version (e.g. release tag, such as `v1.1.0`, or commit hash, such as `26cc08f`) you have used, and whether you have made any modifications to the code.

Read and understand the file `LICENSE`; in particular, clause 7 ("Disclaimer of Warranty"), which states:

> Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

4.1.2 2. Cite the scientific publication

Cite the following manuscript:

> Daniel Huppmann, Matthew Gidden, Oliver Fricko, Peter Kolp, Clara Orthofer, Michael Pimmer, Nikolay Kushin, Adriano Vinca, Alessio Mastrucci, Keywan Riahi, and Volker Krey.
> "The MESSAGEix Integrated Assessment Model and the ix modeling platform".
> *Environmental Modelling & Software* 112:143-156, 2019.
> doi: 10.1016/j.envsoft.2018.11.012
> electronic pre-print available at pure.iiasa.ac.at/15157/.

In addition, include a hyperlink to the online resource MESSAGEix.iiasa.ac.at.

4.1.3 3. Use the naming convention for new model instances

For any new model instance under the MESSAGE*ix* framework, choose a name of the form "MESSAGEix [xxx]" or "MESSAGEix-[xxx]", where [xxx] is replaced by:

- the institution or organization developing the model,
- the name of a country/region represented in the model, or
- a similar identifier.

For example, the national model for South Africa developed by Orthofer et al. [1] is named "MESSAGEix South Africa".

Ensure there is no naming conflict with existing versions of the MESSAGE*ix* model family. When in doubt, contact the IIASA Energy Program at <message_ix@iiasa.ac.at> for a list of existing model instances.

4.1.4 4. Give notice of publication

E-mail <message_ix@iiasa.ac.at> with notice of notice of any new or pending publication.

4.1.5 References

[1] Clara Orthofer, Daniel Huppmann, and Volker Krey (2019).
"South Africa's shale gas resources - chance or challenge?"
Frontiers in Energy Research 7:20. doi: 10.3389/fenrg.2019.00020

4.2 Contributing to MESSAGE*ix* development

We appreciate contributions to the code base and development of new features for the framework. Please open a new issue on Github to raise questions concerning potential bugs or to propose new features. Existing open and resolved/closed issues may already contain a solution.

For contributions to the code base of the platform, please open a GitHub "pull request," including a detailed description of the new feature and unit tests to illustrate the intended functionality. All pull requests will be reviewed by the message_ix maintainers and/or contributors.

Contributors are required to sign the Contributor License Agreement before any pull request can be reviewed. This ensures that all future users can benefit from your contribution, and that your contributions do not infringe on anyone else's rights. The electronic signature is collected via the cla-assistant when issuing the pull request.

4.2.1 Coding style

Code submitted via pull requests must adhere to the following style:

- Python: follow PEP 8.

- R: follow the style of the existing code base.

- Jupyter notebooks (`.ipynb`):

    - Commit 'bare' notebooks, with no cell output. Notebooks will be run and rendered when the documentation is generated.

- Documentation (`.rst`, `.md`):

    - Do not hard-wrap lines. Start each sentence on a new line.

- Other (file names, CLI, etc.): follow the style of the existing code base.

4.3 Contributor License Agreement

4.3.1 Summary and scope

It may seem self-evident that contributing to a project distributed under an open-source license is an implicit permission to anyone for using the contributed code. However, a formal Contributor License Agreements (CLA) makes contribution terms explicit and provides the project maintainers a record of your agreement to those terms.

A wide range of terms exist in other CLAs, including waiver of moral rights, consequential damages waiver, as-is disclaimer, etc. For this project, we follow the more bare-boned GitHub CLA, which focuses on the three most important clauses: copyright, patent, and source of contribution.

In short, by signing this Contributor License Agreement, you confirm that:

1. Anyone can use your contributions anywhere, for free, forever.

2. Your contributions do not infringe on anyone else's rights.

4.3.2 Definition of terms

The following terms are used throughout this agreement:

- **You** - the person or legal entity including its affiliates asked to accept this agreement. An affiliate is any entity that controls or is controlled by the legal entity, or is under common control with it.

- **Project** - the repositories `message_ix` and `ixmp`, and any derived repositories, projects, or software/code packages.

- **Contribution** - any type of work that is submitted to a Project, including any modifications or additions to existing work.

- **Submitted** - conveyed to a Project via a pull request, commit, issue, or any form of electronic, written, or verbal communication with GitHub, contributors or maintainers.

4.3.3 1. Grant of Copyright License

Subject to the terms and conditions of this agreement, You grant to the Projects' maintainers, contributors and users a perpetual, worldwide, unlimited in scope, non-exclusive, no-charge, royalty-free, irrevocable copyright license to,

in particular without being limited to, reproduce, prepare derivative works of, publicly display, make available, sublicense, and distribute Your contributions and such derivative works in whole or in part. Except for this license, You reserve all moral rights, title, and interest in your contributions.

4.3.4 2. Grant of Patent License

Subject to the terms and conditions of this agreement, You grant to the Projects' maintainers, contributors and users a perpetual, worldwide, unlimited in scope, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer your contributions, in whole or in part, where such license applies only to those patent claims licensable by you that are necessarily infringed by your contribution or by combination of your contribution with the project to which this contribution was submitted.

If any entity institutes patent litigation - including cross-claim or counterclaim in a lawsuit - against You alleging that your contribution or any project it was submitted to constitutes or is responsible for direct or contributory patent infringement, then any patent licenses granted to that entity under this agreement shall terminate as of the date such litigation is filed.

4.3.5 3. Source of Contribution

Your contribution is either your original creation or based upon previous work that, to the best of your knowledge, is covered under an appropriate open source license. You assure that you are legally entitled to submit your contribution and grant the above license, or you have clearly identified the source of the contribution and any license or other restriction (like related patents, trademarks, and license agreements) of which you are personally aware. If your employer(s) or employee(s) have rights to intellectual property that you create, you represent that you have received permission to make the contributions on behalf of that employer/employee, or that your employer/employee has waived such rights for your contributions.

Should the licensor be held responsible for any violation of intellectual property right in relation to your contribution, you shall be fully liable for damages that may arise.

4.3.6 Reference and License

This Contributor License Agreement and the introductory text is adapted from the GitHub Contributor License Agreement, Version 298f3afd updated August 9, 2017. GitHub granted a CC-BY-4.0 License to IIASA to use and modify the text of the CLA.

4.4 Frequently asked questions

4.4.1 What's included in a 'typical' MESSAGE*ix* model?

A typical MESSAGE*ix* model instance is based on a suite of technologies such as power plants, which represent a reference energy system (RES). Each technology is characterised by its input and output commodities, costs (investment, fixed and variable components), and other technical/engineering parameters. The model minimizes the total system cost while meeting a given demand for energy services or commodities.

4.4.2 Which policies and regulatory measures can be included?

The MESSAGE*ix* framework can represent a wide range of mitigation options and policies to analyse transformation pathways. For example, bounds or taxes on emissions can be defined to shift the system towards a low-emission technology mix. Upper or lower bounds on deployment of new technologies can also be easily included.

4.4.3 Answered elsewhere

- Under which license is MESSAGE*ix* released? → see `README.md` or the *the documentation index*.

- Can I use MESSAGE*ix* for my own analysis? → see *User guidelines and notice*.

- How can I contribute to the development of the MESSAGE*ix* framework? → see *Contributing to MESSAGEix development*.

- I have a question not answered here → see *the documentation index* for information on the community mailing list.

4.5 References

Bibliography

[1] Daniel Huppmann, Matthew Gidden, Oliver Fricko, Peter Kolp, Clara Orthofer, Michael Pimmer, Nikolay Kushin, Adriano Vinca, Alessio Mastrucci, Keywan Riahi, and Volker Krey. The MESSAGEix Integrated Assessment Model and the ix modeling platform (ixmp): An open framework for integrated and cross-cutting analysis of energy, climate, the environment, and sustainable development. *Environmental Modelling & Software*, 112:143–156, 2019. doi:10.1016/j.envsoft.2018.11.012.

[2] Nils Johnson, Manfred Strubegger, Madeleine McPherson, Simon C. Parkinson, Volker Krey, and Patrick Sullivan. A reduced-form approach for representing the impacts of wind and solar PV deployment on the structure and operation of the electricity system. *Energy Economics*, 64:651–664, 2016. doi:10.1016/j.eneco.2016.07.010.

[3] Ilkka Keppo and Manfred Strubegger. Short term decisions for long term problems – The effect of foresight on model based energy systems analysis. *Energy*, 35(5):2033–2042, 2010. doi:10.1016/j.energy.2010.01.019.

[4] Alan Sussmann Manne and Richard G Richels. *Buying greenhouse insurance: the economic costs of carbon dioxide emission limits*. MIT press, 1992. ISBN 0-262-13280-X.

[5] Sabine Messner and Manfred Strubegger. User's Guide for MESSAGE III. 1995. URL: http://webarchive.iiasa.ac.at/Admin/PUB/Documents/WP-95-069.pdf.

[6] Patrick Sullivan, Volker Krey, and Keywan Riahi. Impacts of considering electric sector variability and reliability in the MESSAGE model. *Energy Strategy Reviews*, 1(3):157 – 163, 2013. doi:10.1016/j.esr.2013.01.001.

Python Module Index

i

m

Index