message Documentation

Release 2.0.0

IIASA Energy Program

Jan 14, 2020

Contents

1 Overview and scope **3**

2 Getting Started **5**
2.1 Installation 5
2.2 Tutorials 7

3 Detailed documentation **9**
3.1 MESSAGE*ix* framework overview 9
3.2 Python & R API 12
3.3 Mathematical specification 25
3.4 Developing MESSAGE*ix* models 57

4 Using and contributing to MESSAGE*ix* **79**
4.1 What's New 79
4.2 User guidelines and notice 83
4.3 Contributing to MESSAGE*ix* development 84
4.4 Contributor License Agreement 89
4.5 Frequently asked questions 90
4.6 References 90

Bibliography **91**

Python Module Index **93**

Index **95**

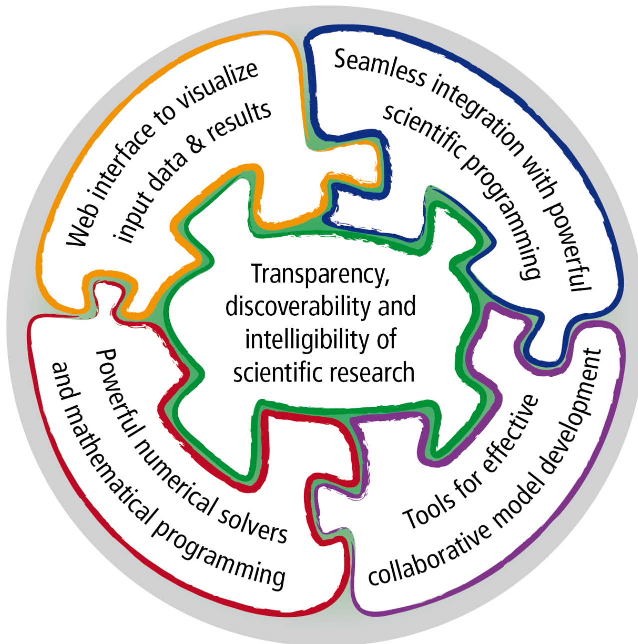

Fig. 1: The *ix modeling platform* (source: [1])

CHAPTER 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.

CHAPTER 2

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.

   Note: MESSAGE-MACRO requires GAMS 24.8.1 or later (see *MESSAGE_MACRO.GAMS_min_version*).

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` (macOS) 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 --editable .
```

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

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

2.1.4 Common issues

No JVM shared library file (jvm.dll) found

If you get an error containing "No JVM shared library file (jvm.dll) found" when creating a `Platform` object (e.g. `mp = ix.Platform(driver='HSQLDB')`), it is likely that you need to set the `JAVA_HOME` environment variable (see for example these instructions).

2.1.5 Copy GAMS model files for editing

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 `message-ix` command-line program to copy the model files in a directory of your choice:

```
$ message-ix copy-model /path/for/model/files
```

You can also set the `message model dir` configuration key so that this copy of the files is used by default:

```
$ message-ix config set "message model dir" /path/for/model/files
```

. . . or do both in one step:

```
$ message-ix copy-model --set-default /path/for/model/files
```

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 `message-ix` command-line program. In a command prompt:

```
$ message-ix dl /path/to/tutorials
```

Note: By default, the tutorials for your installed version of MESSAGE*ix* are downloaded. To download a different version, add e.g. `--tag 1.2.0` to the above command. To download the tutorials from the development version, add `--branch master`.

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 `message-ix 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 (`westeros_baseline.ipynb`).
2. Add extra detail and constraints to the model
    1. Emissions
        1. Introduce emissions and a bound on the emissions (*westeros_emissions_bounds.ipynb*).
        2. Introducing taxes on emissions (*westeros_emissions_taxes.ipynb*).
    2. 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 (`westeros_firm_capacity.ipynb`).
    3. Represent coal and wind electricity using a different, “flexibility requirement” formulation, wherein wind *requires* and coal *supplies* flexibility (`westeros_flexible_generation.ipynb`).
    4. Variablity in energy supply and demand, by adding sub-annual time steps - winter and summer (`westeros_seasonality.ipynb`).
3. Post-processing: learn how to use ixmp and message_ix reporting features _after_ the MESSAGE model has run (`westeros_report.ipynb`).

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>.

CHAPTER 3

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:

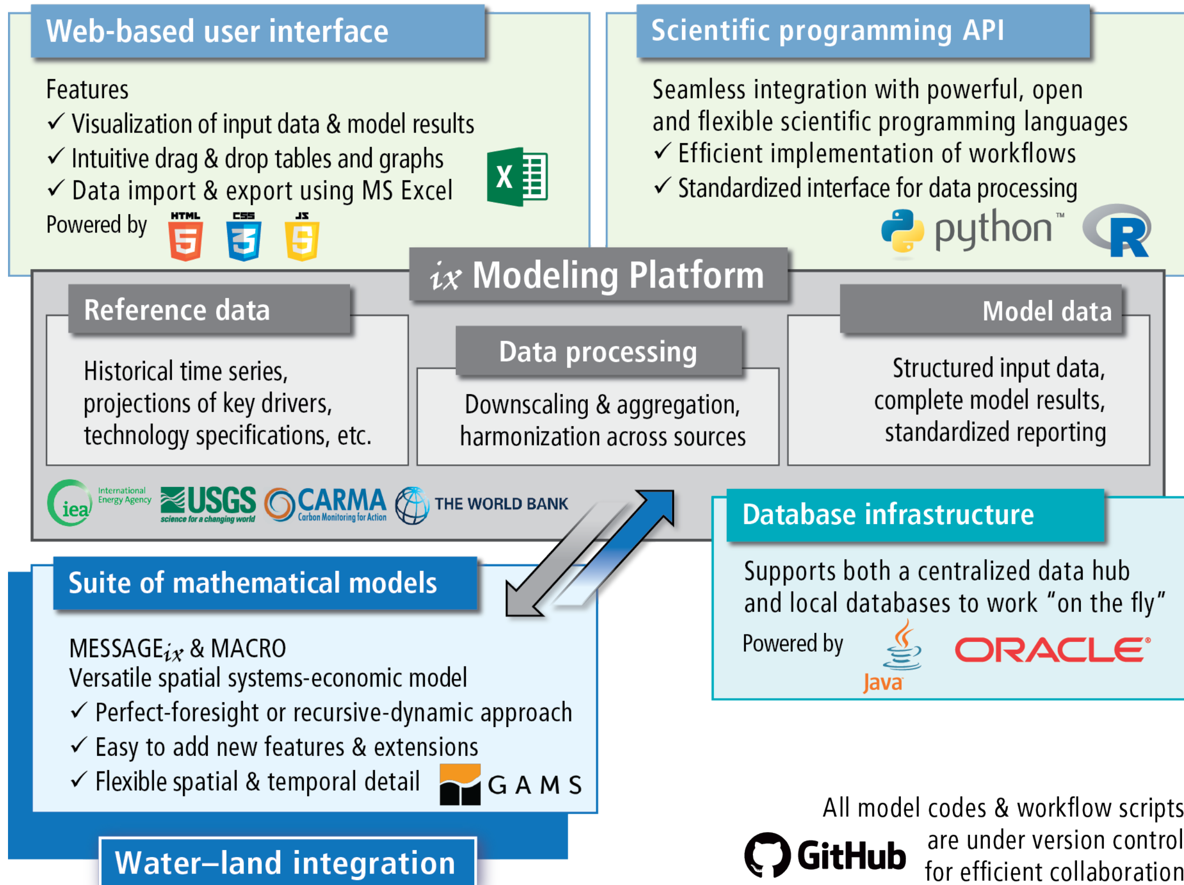

Fig. 1: Components and their interlinkages in the *ix modeling platform* (source [1]): web-based user interface, scientific programming interface, modeling platform, database backend, implementation of the MESSAGE*ix* mathematical model formulation.

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 2.0.0. 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 2.0.0. 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:

- *ixmp package*
- *message_ix package*
- *Model classes*
- *Utility methods*
- *Testing utilities*

Support for R usage of the core classes is provided through the reticulate package. For instance:

```
> library(reticulate)
> ixmp <- import('ixmp')
> message_ix <- import('message_ix')
> mp <- ixmp$Platform(...)
> scen <- message_ix$Scenario(mp, ...)
```

3.2.1 ixmp package

`ixmp` provides three classes. These are fully described by the ixmp documentation, which is cross-linked from many places in the MESSAGE*ix* documentation.

`Platform`(*args[, name, backend])	Instance of the modeling platform.
`TimeSeries`(mp, model, scenario[, version, . . .])	Collection of data in time series format.
`Scenario`(mp, model, scenario[, version, . . .])	Collection of model-related data.

`ixmp` also provides some utility classes and methods:

`ixmp.config`	Configuration for ixmp.
`ixmp.model.MODELS`	Mapping from names to available models.
`ixmp.model.get_model`(name, **model_options)	Return a model for *name* (or the default) with *model_options*.
`ixmp.testing.make_dantzig`	

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`.

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

Bases: `ixmp.core.Scenario`

MESSAGE*ix* Scenario.

See `ixmp.TimeSeries` for the meaning of arguments *mp*, *model*, *scenario*, *version*, and *annotation*; `ixmp.Scenario` for the meaning of *cache*. The *scheme* of a newly-created Scenario is always 'MESSAGE'.

This class extends `ixmp.Scenario` and `ixmp.TimeSeries` and inherits all their methods. Documentation of these inherited methods is included here for convenience. `message_ix.Scenario` defines additional methods specific to MESSAGE*ix*:

`add_cat`(name, cat, keys[, is_unique])	Map elements from *keys* to category *cat* within set *name*.
`add_horizon`(data)	Add sets related to temporal dimensions of the model.
`add_spatial_sets`(data)	Add sets related to spatial dimensions of the model.
`cat`(name, cat)	Return a list of all set elements mapped to a category.
`cat_list`(name)	Return a list of all categories for a mapping set.
`equ`(name[, filters])	Return equation data.
`firstmodelyear`	The first model year of the scenario.
`par`(name[, filters])	Return parameter data.
`read_excel`(fname[, add_units, commit_steps])	Read Excel file data and load into the scenario.
`rename`(name, mapping[, keep])	Rename an element in a set
`to_excel`(fname)	Save a scenario as an Excel file.
`var`(name[, filters])	Return variable data.
`vintage_and_active_years`([ya_args, in_horizon])	Return sets of vintage and active years for use in data input.
`years_active`(node, tec, yr_vtg)	Return years in which *tec* of *yr_vtg* can be active in *node*.

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_geodata(*df*)

Add geodata (layers) to the TimeSeries.

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

- *region*
- *variable*

- *time*
- *unit*
- *year*
- *value*
- *meta*

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_par (*name*, *key_or_data=None*, *value=None*, *unit=None*, *comment=None*)

Set the values of a parameter.

Parameters

- **name** (*str*) – Name of the parameter.
- **key_or_data** (*str or iterable of str or range or dict or pandas.DataFrame*) – Element(s) to be added.
- **value** (*numeric or iterable of numeric, optional*) – Values.
- **unit** (*str or iterable of str, optional*) – Unit symbols.
- **comment** (*str or iterable of str, optional*) – Comment(s) for the added values.

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). If given, there must be the same number of comments as elements.

Raises

- KeyError – If the set *name* does not exist. *init_set()* must be called before *add_set()*.
- ValueError – For invalid forms or combinations of *key* and *comment*.

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']}}})
```

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'`.

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

Parameters

- **name** (*str*) – Name of the set.
- **cat** (*str*) – Name of the category.

Returns

Return type list of str

cat_list (*name*)
Return a list of all categories for a mapping set.

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

check_out(*timeseries_only=False*)
Check out the TimeSeries for modification.

clone(**args*, ***kwargs*)
Clone the current scenario and return the clone.

See `ixmp.Scenario.clone()` for other parameters.

Parameters

- **keep_solution** (*bool, optional*) – 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()`.

commit(*comment*)
Commit all changed data to the database.

If the TimeSeries was newly created (with `version='new'`), `version` is updated with a new version number assigned by the backend. Otherwise, `commit()` does not change the `version`.

Parameters **comment** (*str*) – Description of the changes being committed.

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

equ(*name*, *filters=None*)
Return equation data.

Same as `ixmp.Scenario.equ()`, except columns indexed by the MESSAGE*ix* set `year` are returned with `int` dtype.

Parameters

- **name** (*str*) – Name of the equation.
- **filters** (*dict (str -> list of str), optional*) – Filters for the dimensions of the equation.

Returns Filtered elements of the equation.

Return type pd.DataFrame

equ_list()
List all defined equations.

firstmodelyear
The first model year of the scenario.

Returns

Return type int

classmethod from_url(*url*, *errors='warn'*)
Instantiate a Scenario given an ixmp-scheme URL.

The following are equivalent:

```
from ixmp import Platform, Scenario
mp = Platform(name='example')
scen = Scenario(mp 'model', 'scenario', version=42)
```

and:

```
from ixmp import Scenario
scen, mp = Scenario.from_url('ixmp://example/model/scenario#42')
```

Parameters

- **url** (*str*) – See parse_url.
- **errors** (*'warn' or 'raise'*) – If 'warn', a failure to load the Scenario is logged as a warning, and the platform is still returned. If 'raise', the exception is raised.

Returns **scenario, platform** – The Scenario and Platform referred to by the URL.

Return type 2-tuple of (Scenario, Platform)

get_geodata()
Fetch geodata and return it as dataframe.

Returns Specified data.

Return type pandas.DataFrame

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 *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

- `ValueError` – If the set (or another object with the same *name*) already exists.
- `RuntimeError` – If the Scenario is not checked out (see `check_out()`).

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

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

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

load_scenario_data()
Load all Scenario data into memory.

Raises `ValueError` – If the Scenario was instantiated with `cache=False`.

par(*name*, *filters=None*)
Return parameter data.

Same as `ixmp.Scenario.par()`, except columns indexed by the MESSAGE*ix* set `year` are returned with `int` dtype.

Parameters

- **name** (*str*) – Name of the parameter.
- **filters** (*dict (str -> list of str), optional*) – Filters for the dimensions of the parameter.

Returns Filtered elements of the parameter.

Return type pd.DataFrame

par_list()
List all defined parameters.

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

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_geodata(*df*)
Remove geodata from the TimeSeries instance.

Parameters df (`pandas.DataFrame`) – Data to remove. *df* must have the following columns:

- *region*
- *variable*
- *unit*
- *time*
- *year*

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*.

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*

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

run_id ()
get the run id of this TimeSeries

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*)
Return elements of a set.

Same as `ixmp.Scenario.set()`, except columns for multi-dimensional sets indexed by the MESSAGE*ix* set `year` are returned with `int` dtype.

Parameters

- **name** (*str*) – Name of the set.
- **filters** (*dict (str -> list of str), optional*) – 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

- *pd.Series* – If *name* is an index set.
- *pd.DataFrame* – If *name* is a set defined over one or more other, index sets.

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

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='MESSAGE'*, *solve_options={}*, ***kwargs*)
Solve MESSAGE or MESSAGE-MACRO for the Scenario.

By default, `ixmp.Scenario.solve()` is called with 'MESSAGE' as the *model* argument. *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 (str -> str), optional*) – Name to value mapping to use for GAMS CPLEX solver options file. See *MESSAGE* and *DEFAULT_CPLEX_OPTIONS*.
- **kwargs** – Many other options control the execution of the underlying GAMS code; see `GAMSModel`.

timeseries(*region=None*, *variable=None*, *unit=None*, *year=None*, *iamc=False*)
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`

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

var (*name*, *filters=None*)
Return variable data.

Same as `ixmp.Scenario.var()`, except columns indexed by the MESSAGE*ix* set `year` are returned with `int` dtype.

Parameters

- **name** (*str*) – Name of the variable.
- **filters** (*dict (str -> list of str), optional*) – Filters for the dimensions of the variable.

Returns Filtered elements of the variable.

Return type pd.DataFrame

var_list ()
List all defined variables.

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, tec, yr_vtg), optional*) – Arguments to *years_active()*.
- **in_horizon** (*bool, optional*) – Only return years within the model horizon (*firstmodelyear* or later).

Returns with columns 'year_vtg' and 'year_act', in which each row is a valid pair.

Return type pandas.DataFrame

years_active (*node*, *tec*, *yr_vtg*)
Return years in which *tec* of *yr_vtg* can be active in *node*.

The *parameters* `duration_period` and `technical_lifetime` are used to determine which periods are partly or fully within the lifetime of the technology.

Parameters

- **node** (*str*) – Node name.
- **tec** (*str*) – Technology name.
- **yr_vtg** (*int or str*) – Vintage year.

Returns

Return type list of int

3.2.3 Model classes

message_ix.models.**DEFAULT_CPLEX_OPTIONS = {'advind': 0, 'epopt': 1e-06, 'lpmethod': 2,**

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.models.**MESSAGE**(*name=None*, ***model_options*)

Bases: ixmp.model.gams.GAMSModel

The MESSAGE Python class encapsulates the GAMS code for the core MESSAGE mathematical formulation. The *model_options* arguments are received from *Scenario.solve()*, and—except for *solve_options*—are passed on to the parent class GAMSModel; see there for a full list of options.

name = 'MESSAGE'

defaults = dict(...)

Default model options. The paths to MESSAGE GAMS source files use the MODEL_PATH configuration setting. MODEL_PATH, in turn, defaults to "message_ix/model" inside the directory where message_ix is installed.

Key	Value
MESSAGE defaults	
model_file	'{MODEL_PATH}/{model_name}_run.gms'
in_file	'{MODEL_PATH}/data/MsgData_{case}.gdx'
out_file	'{MODEL_PATH}/output/MsgOutput_{case}.gdx'
solve_args	['--in="{in_file}"', '--out="{out_file}"', '--iter="{MODEL_PATH}/output/MsgIterationReport_{case}.gdx"']
Inherited from GAMSModel	
case	'{scenario.model}_{scenario.scenario}'
gams_args	['LogOption=4']
check_solution	True
comment	None
equ_list	None
var_list	None

classmethod read_version()

Retrieve MESSAGE version string from version.gms.

run(*scenario*)

Execute the model.

MESSAGE creates a file named cplex.opt in the model directory, containing the options in *DEFAULT_CPLEX_OPTIONS*, or any overrides passed to *solve()*.

class message_ix.models.**MESSAGE_MACRO**(**args*, ***kwargs*)

Bases: *message_ix.models.MESSAGE*

name = 'MESSAGE-MACRO'

GAMS_min_version = '24.8.1'

MESSAGE-MACRO uses the GAMS break; statement, and thus requires GAMS 24.8.1 or later.

3.2.4 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
```

3.2.5 Testing utilities

message_ix.testing.**make_dantzig**(*mp*, *solve=False*, *multi_year=False*, ***solve_opts*)

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][3][4]	$y \in Y$	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)

[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.

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.

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 COMMODITY_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)

[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 aggregrating over a specific technology type. The element "all" is added by default in a `MESSAGE`-scheme `message_ix`.Scenario.

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'.

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

In MESSAGE*ix*, the volume of resources at the start of the model horizon is defined by `resource_volume`. The quantity of the resources that are extracted per year is dependent on two parameters. The first is `bound_extraction_up`, which constraints the maximum extraction of the resources (by grade) in a year. The

[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.

second is `resource_remaining`, which is the maximum extraction of the remaining resources in a certain year, as a percentage. Extraction costs for resources are represented by `resource_cost` parameter.

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`

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

[5] Commodity stock refers to an exogenous (initial) quantity of commodity in stock. 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.

[7] The parameter `demand` in a `MESSAGE`-scheme `ixmp`.Scenario is translated to the parameter `demand_fixed` in the MESSAGE*ix* 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` (maximum peak load factor for reliability constraint of firm capacity) and `reliability_factor` (reliability of a technology (per rating)) are based on the formulation proposed by Sullivan et al., 2013 [6]. It is used in *Reliability of installed capacity*.

Input/output mapping, costs and engineering specifications

Parameter name	Index dimensions
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[11]	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[13]	node_loc | technology | year_vtg | year_act | mode | commodity | level | time | rating
renewable_capacity_factor[14]	node_loc | commodity | grade | level | year
renewable_potential[14]	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. The bounds on activity are implemented as the aggregate over all vintages in a specific period (*Equation ACTIVITY_BOUND_UP* and *Equation ACTIVITY_BOUND_LO*).

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

[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.

[11] The construction time only has an effect on the investment costs; in MESSAGE*ix*, each unit of new-built capacity is available instantaneously at the beginning of the model period.

[12] Maximum share of technology in commodity use per rating. 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.

[13] Contribution of technologies towards operation flexibility constraint. It is used in *Equation SYSTEM_FLEXIBILITY_CONSTRAINT*.

[14] `renewable_capacity_factor` refers to the quality of renewable potential by grade and `renewable_potential` refers to the size of the renewable potential per grade.

Dynamic constraints on capacity and activity

The following parameters specify constraints on the growth of new capacity and activity, i.e., market penetration. 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]). For more information, please refer to the equations in section *Dynamic constraints on market penetration* of the mathematical formulation.

Parameter name	Index names
initial_new_capacity_up	`node_loc	tec	year_vtg`
growth_new_capacity_up[15]	`node_loc	tec	year_vtg`
soft_new_capacity_up[15]	`node_loc	tec	year_vtg`
initial_new_capacity_lo	`node_loc	tec	year_vtg`
growth_new_capacity_lo[15]	`node_loc	tec_actual	year_vtg`
soft_new_capacity_lo[15]	`node_loc	tec	year_vtg`
initial_activity_up[16]	`node_loc	tec	year_act	time`
growth_activity_up[15][16]	`node_loc	tec	year_act	time`
soft_activity_up[15]	`node_loc	tec	year_act	time`
initial_activity_lo[16]	`node_loc	tec	year_act	time`
growth_activity_lo[15][16]	`node_loc	tec	year_act	time`
soft_activity_lo[15]	`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). Upper and lower bounds of add-on technologies are defined relative to the parent: `addon_up` and `addon_lo`, respectively.

Note: No default `addon_conversion` factor (conversion factor between add-on and parent technology activity) is set. This is to avoid default conversion factors of 1 being set for technologies with multiple 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`

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*. Absolute cost and levelized cost multipliers are used for the relaxation of upper and lower bounds.

[15] 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).

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

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

Auxiliary investment cost parameters include the remaining technical lifetime at the end of model horizon (beyond_horizon_lifetime) in addition to the different scaling factors and multipliers as listed below. These factors account for remaining capacity (remaining_capacity) or construction time of new capacity (construction_time_factor), the value of investment at the end of model horizon (end_of_horizon_factor) or the discount factor of remaining lifetime beyond model horizon (beyond_horizon_factor).

Parameter name	Index names
construction_time_factor	node | tec | year_all
remaining_capacity	node | tec | year_all
end_of_horizon_factor	node | tec | year_all
beyond_horizon_lifetime	node | tec | year_all
beyond_horizon_factor	node | tec | year_all

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[17]	type_emission | emission
bound_emission	node | type_emission | type_tec | type_year
tax_emission	node | type_emission | type_tec | type_year

[17] The parameter emission_scaling is the scaling factor to harmonize bounds or taxes across types of emissions. It allows to efficiently aggregate different emissions/pollutants and set bounds or taxes on various categories.

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. The parameters listed below refer to the assigned land scenario.

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/mode to be active on a certain level. For the mathematical formulation, refer to *Constraints on shares of technologies and commodities*.

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*. For the formulation of the relations, refer to *Section of generic relations (linear constraints)*.

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 mimic 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

$$OBJ = \sum_{n,y \in Y^M} df_year_y \cdot COST_NODAL_{n,y}$$

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.

$$
\begin{aligned}
COST_NODAL_{n,y} = & \sum_{c,g} resource_cost_{n,c,g,y} \cdot EXT_{n,c,g,y} \\
& + \sum_{t} \Big(inv_cost_{n,t,y} \cdot construction_time_factor_{n,t,y} \\
& \quad \cdot end_of_horizon_factor_{n,t,y} \cdot CAP_NEW_{n,t,y} \\
& \quad + \sum_{y^V \leq y} fix_cost_{n,t,y^V,y} \cdot CAP_{n,t,y^V,y} \\
& \quad + \sum_{\substack{y^V \leq y \\ m,h}} var_cost_{n,t,y^V,y,m,h} \cdot ACT_{n,t,y^V,y,m,h} \\
& \quad + \Big(abs_cost_new_capacity_soft_up_{n,t,y} \\
& \quad + level_cost_new_capacity_soft_up_{n,t,y} \cdot inv_cost_{n,t,y} \Big) \cdot CAP_NEW_UP_{n,t,y} \\
& \quad + \Big(abs_cost_new_capacity_soft_lo_{n,t,y} \\
& \quad + level_cost_new_capacity_soft_lo_{n,t,y} \cdot inv_cost_{n,t,y} \Big) \cdot CAP_NEW_LO_{n,t,y} \\
& \quad + \sum_{m,h} \Big(abs_cost_activity_soft_up_{n,t,y,m,h} \\
& \quad + level_cost_activity_soft_up_{n,t,y,m,h} \cdot levelized_cost_{n,t,y,m,h} \Big) \cdot ACT_UP_{n,t,y,h} \\
& \quad + \sum_{m,h} \Big(abs_cost_activity_soft_lo_{n,t,y,m,h} \\
& \quad + level_cost_activity_soft_lo_{n,t,y,m,h} \cdot levelized_cost_{n,t,y,m,h} \Big) \cdot ACT_LO_{n,t,y,h} \Big) \\
& + \sum_{\substack{\widehat{e},\widehat{t} \\ e \in E(\widehat{e})}} emission_scaling_{\widehat{e},e} \cdot emission_tax_{n,\widehat{e},\widehat{t},y} \cdot EMISS_{n,e,\widehat{t},y} \\
& + \sum_{s} land_cost_{n,s,y} \cdot LAND_{n,s,y} \\
& + \sum_{r} relation_cost_{r,n,y} \cdot REL_{r,n,y}
\end{aligned}
$$

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.

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

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.

$$EXT_{n,c,g,y} \leq bound_extraction_up_{n,c,g,y}$$

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.

$$EXT_{n,c,g,y} \leq resource_remaining_{n,c,g,y} \cdot \Big(resource_volume_{n,c,g} - \sum_{y' < y} duration_period_{y'} \cdot EXT_{n,c,g,y'} \Big)$$

Equation RESOURCE_HORIZON

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

$$\sum_{y} duration_period_{y} \cdot EXT_{n,c,g,y} \leq resource_volume_{n,c,g}$$

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.

$$\begin{aligned} CAP_{n,t,y^V,'first_period'} \leq remaining_capacity_{n,t,y^V,'first_period'} \cdot duration_period_{y^V} \cdot historical_new_capacity_{n,t,y^V} \\ \text{if } y^V <' first_period' \text{ and } |y| - |y^V| < technical_lifetime_{n,t,y^V} \quad \forall \, t \in T^{INV} \end{aligned}$$

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).

$$CAP_{n,t,y^V,y^V} = remaining_capacity_{n,t,y^V,y^V} \cdot duration_period_{y^V} \cdot CAP_NEWn,t,y^V \quad \forall \, t \in T^{INV}$$

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.

$$\begin{aligned} CAP_{n,t,y^V,y} \leq remaining_capacity_{n,t,y^V,y} \cdot CAP_{n,t,y^V,y-1} \\ \text{if } y > y^V \text{ and } y^V >' first_period' \text{ and } |y| - |y^V| < technical_lifetime_{n,t,y^V} \quad \forall \, t \in T^{INV} \end{aligned}$$

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].

$$\sum_{\substack{t,q^{t \in T^{INV}} \\ y^V \leq y}} reliability_factor_{n,t,y,c,l,h,'firm'} \cdot CAP_FIRM_{n,t,c,l,y}$$
$$+ \sum_{t,q,y^V \leq y} reliability_factor_{n,t,y,c,l,h,q} \cdot ACT_RATING_{n,t,y^V,y,c,l,h,q}$$
$$\geq peak_load_factor_{n,c,l,y,h} \cdot COMMODITY_USE_{n,c,l,y}$$

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

$$\begin{aligned}
&\sum_{\substack{n^L,t,m,h^A \\ y^V \leq y, (n,\widehat{t},m,c,l) \sim S^{share}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h}) \\
&\quad \cdot duration_time_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A} \\
&\geq share_commodity_up_{s,n,y,h} \cdot \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y, (n,\widehat{t},m,c,l) \sim S^{total}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h}) \\
&\quad \cdot duration_time_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}
\end{aligned}$$

This constraint is only active if $share_commodity_up_{s,n,y,h}$ is defined.

Equation SHARE_CONSTRAINT_COMMODITY_LO

$$\begin{aligned}
&\sum_{\substack{n^L,t,m,h^A \\ y^V \leq y, (n,\widehat{t},m,c,l) \sim S^{share}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h}) \\
&\quad \cdot duration_time_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A} \\
&\leq share_commodity_lo_{s,n,y,h} \cdot \sum_{\substack{n^L,t,m,h^A \\ y^V \leq y, (n,\widehat{t},m,c,l) \sim S^{total}}} (output_{n^L,t,y^V,y,m,n,c,l,h^A,h} + input_{n^L,t,y^V,y,m,n,c,l,h^A,h}) \\
&\quad \cdot duration_time_rel_{h,h^A} \cdot ACT_{n^L,t,y^V,y,m,h^A}
\end{aligned}$$

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.

$$
\begin{aligned}
CAP_NEW_{n,t,y} \leq & \; initial_new_capacity_up_{n,t,y} \cdot \frac{ \Big(1 + growth_new_capacity_up_{n,t,y} \Big)^{|y|} - 1 }{ growth_new_capacity_up_{n,t,y} } \\
& + \Big(CAP_NEW_{n,t,y-1} + historical_new_capacity_{n,t,y-1} \Big) \\
& \hspace{2 cm} \cdot \Big(1 + growth_new_capacity_up_{n,t,y} \Big)^{|y|} \\
& + CAP_NEW_UP_{n,t,y} \cdot \Bigg(\Big(1 + soft_new_capacity_up_{n,t,y}\Big)^{|y|} - 1 \Bigg) \\
& \quad \forall \ t \ \in \ T^{INV}
\end{aligned}
$$

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.

$$
\begin{aligned}
CAP_NEW_{n,t,y} \geq & - initial_new_capacity_lo_{n,t,y} \cdot \frac{ \Big(1 + growth_new_capacity_lo_{n,t,y} \Big)^{|y|} }{ growth_new_capacity_lo_{n,t,y} } \\
& + \Big(CAP_NEW_{n,t,y-1} + historical_new_capacity_{n,t,y-1} \Big) \\
& \hspace{2 cm} \cdot \Big(1 + growth_new_capacity_lo_{n,t,y} \Big)^{|y|} \\
& - CAP_NEW_LO_{n,t,y} \cdot \Bigg(\Big(1 + soft_new_capacity_lo_{n,t,y} \Big)^{|y|} - 1 \Bigg) \\
& \quad \forall \ t \ \in \ T^{INV}
\end{aligned}
$$

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.

$$
\begin{aligned}
\sum_{y^V \leq y, m} ACT_{n,t,y^V,y,m,h} \leq\; & initial_activity_up_{n,t,y,h} \cdot \frac{\left(1 + growth_activity_up_{n,t,y,h}\right)^{|y|} - 1}{growth_activity_up_{n,t,y,h}} \\
& + \left(\sum_{y^V \leq y-1, m} ACT_{n,t,y^V,y-1,m,h} + \sum_{m} historical_activity_{n,t,y-1,m,h} \right) \\
& \cdot \left(1 + growth_activity_up_{n,t,y,h}\right)^{|y|} \\
& + ACT_UP_{n,t,y,h} \cdot \left(\left(1 + soft_activity_up_{n,t,y,h}\right)^{|y|} - 1 \right)
\end{aligned}
$$

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.

$$
\begin{aligned}
\sum_{y^V \leq y, m} ACT_{n,t,y^V,y,m,h} \geq\; & - initial_activity_lo_{n,t,y,h} \cdot \frac{\left(1 + growth_activity_lo_{n,t,y,h}\right)^{|y|} - 1}{growth_activity_lo_{n,t,y,h}} \\
& + \left(\sum_{y^V \leq y-1, m} ACT_{n,t,y^V,y-1,m,h} + \sum_{m} historical_activity_{n,t,y-1,m,h} \right) \\
& \cdot \left(1 + growth_activity_lo_{n,t,y,h}\right)^{|y|} \\
& - ACT_LO_{n,t,y,h} \cdot \left(\left(1 + soft_activity_lo_{n,t,y,h}\right)^{|y|} - 1 \right)
\end{aligned}
$$

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.

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

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.

$$\frac{\sum_{y' \in Y(\widehat{y}), e \in E(\widehat{e})} \begin{array}{l} duration_period_{y'} \cdot emission_scaling_{\widehat{e},e} \cdot \\ \left(EMISS_{n,e,\widehat{t},y'} + \sum_m historical_emission_{n,e,\widehat{t},y'} \right) \end{array}}{\sum_{y' \in Y(\widehat{y})} duration_period_{y'}} \leq bound_emission_{n,\widehat{e},\widehat{t},\widehat{y}}$$

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

$$
\begin{aligned}
LAND_{n,s,y} \leq & initial_land_scen_up_{n,s,y} \cdot \frac{\left(1 + growth_land_scen_up_{n,s,y}\right)^{|y|} - 1}{growth_land_scen_up_{n,s,y}} \\
& + \left(LAND_{n,s,y-1} + historical_land_{n,s,y-1}\right) \cdot \left(1 + growth_land_scen_up_{n,s,y}\right)^{|y|}
\end{aligned}
$$

Equation DYNAMIC_LAND_SCEN_CONSTRAINT_LO

$$
\begin{aligned}
LAND_{n,s,y} \geq & - initial_land_scen_lo_{n,s,y} \cdot \frac{\left(1 + growth_land_scen_lo_{n,s,y}\right)^{|y|} - 1}{growth_land_scen_lo_{n,s,y}} \\
& + \left(LAND_{n,s,y-1} + historical_land_{n,s,y-1}\right) \cdot \left(1 + growth_land_scen_lo_{n,s,y}\right)^{|y|}
\end{aligned}
$$

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

$$
\begin{aligned}
\sum_{s \in S} land_use_{n,s,y,u} \cdot LAND_{n,s,y} \leq & \; initial_land_up_{n,y,u} \cdot \frac{\left(1 + growth_land_up_{n,y,u}\right)^{|y|} - 1}{growth_land_up_{n,y,u}} \\
& + \Big(\sum_{s \in S} \left(land_use_{n,s,y-1,u} + dynamic_land_up_{n,s,y-1,u}\right) \\
& \quad \cdot \left(LAND_{n,s,y-1} + historical_land_{n,s,y-1}\right) \Big) \\
& \quad \cdot \left(1 + growth_land_up_{n,y,u}\right)^{|y|}
\end{aligned}
$$

Equation DYNAMIC_LAND_TYPE_CONSTRAINT_LO

$$
\begin{aligned}
\sum_{s \in S} land_use_{n,s,y,u} \cdot LAND_{n,s,y} \geq & -initial_land_lo_{n,y,u} \cdot \frac{\left(1 + growth_land_lo_{n,y,u}\right)^{|y|} - 1}{growth_land_lo_{n,y,u}} \\
& + \Big(\sum_{s \in S} \left(land_use_{n,s,y-1,u} + dynamic_land_lo_{n,s,y-1,u}\right) \\
& \quad \cdot \left(LAND_{n,s,y-1} + historical_land_{n,s,y-1}\right) \Big) \\
& \quad \cdot \left(1 + growth_land_lo_{n,y,u}\right)^{|y|}
\end{aligned}
$$

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

$$
\begin{aligned}
REL_{r,n,y} = \sum_t \Bigg(& relation_new_capacity_{r,n,y,t} \cdot CAP_NEW_{n,t,y} \\
& + relation_total_capacity_{r,n,y,t} \cdot \sum_{y^V \leq y} CAP_{n,t,y^V,y} \\
& + \sum_{n^L,y',m,h} relation_activity_{r,n,y,n^L,t,y',m} \\
& \quad \cdot \Big(\sum_{y^V \leq y'} ACT_{n^L,t,y^V,y',m,h} + historical_activity_{n^L,t,y',m,h} \Big) \Bigg)
\end{aligned}
$$

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 MESSAGEix, 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 2.0. 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!

Contents:

- *Terminology*
- *Basic usage*
- *Customization*
- *Reporters*
- *Computations*

  - *Computations from ixmp*
- *Configuration*
- *Utilities*

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.

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

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.

Reporters

`message_ix.reporting.Reporter`(**kwargs)	MESSAGEix Reporter.
`ixmp.reporting.Reporter`(**kwargs)	Class for generating reports on `ixmp.Scenario` objects.
`ixmp.reporting.Key`(name[, dims, tag])	A hashable key for a quantity that includes its dimensionality.

The `ixmp.Reporter` automatically adds keys based on the contents of the `ixmp.Scenario` argument. The `message_ix.reporting.Reporter` adds additional keys for **derived quantities** specific to the MESSAGEix model framework. These include:

- `out`: the product of `output` (output efficiency) and `ACT` (activity).
- `out_hist`: `output` × `ref_activity` (historical reference activity),
- `in`: `input` × `ACT`,
- `in_hist`: `input` × `ref_activity`,
- `emi`: `emission_factor` × `ACT`,
- `emi_hist`: `emission_factor` × `ref_activity`,
- `inv`: `inv_cost` × `CAP_NEW`,
- `inv_hist`: `inv_cost` × `ref_new_capacity`,
- `fom`: `fix_cost` × `CAP`,
- `fom_hist`: `fix_cost` × `ref_capacity`,
- `vom`: `var_cost` × `ACT`, and
- `vom_hist`: `var_cost` × `ref_activity`.
- `tom`: `fom` + `vom`.

Tip: Use *full_key()* to retrieve the full-dimensionality Key for any of these quantities.

Other added keys include:

- <name>:pyam for the above quantities, plus:
  - CAP:pyam (from CAP)
  - CAP_NEW:pyam (from CAP_NEW)

  These keys return the values in the IAMC data format, as pyam objects.
- Standard reports message:system, message_costs, and message:emissions.
- The report message:default, collecting all of the above reports.

These automatic features of *Reporter* are controlled by:

PRODUCTS	Automatic quantities that are the product() of two others.
DERIVED	Automatic quantities derived by other calculations.
MAPPING_SETS	MESSAGE mapping sets, converted to quantities using *map_as_qty()*.
PYAM_CONVERT	Quantities to automatically convert to IAMC format using *as_pyam()*.
REPORTS	Automatic reports that *concat()* quantities converted to IAMC format.

class message_ix.reporting.**Reporter**(***kwargs*)
Bases: *ixmp.reporting.Reporter*

MESSAGEix Reporter.

convert_pyam(*quantities*, *year_time_dim*, *tag='iamc'*, *drop={}*, *collapse=None*)
Add conversion of one or more **quantities** to IAMC format.

Parameters

- **quantities** (*str or Key or list of (str, Key)*) – Quantities to transform to pyam/IAMC format.
- **year_time_dim** (*str*) – Label of the dimension use for the 'Year' or 'Time' column of the resulting pyam.IamDataFrame. The column is labelled 'Time' if year_time_dim=='h', otherwise 'Year'.
- **tag** (*str, optional*) – Tag to append to new Keys.
- **drop** (*iterable of str, optional*) – Label of additional dimensions to drop from the resulting data frame. Dimensions h, y, ya, yr, and yv— except for the one named by *year_time_dim*—are automatically dropped.
- **collapse** (*callable, optional*) – Callback to handle additional dimensions of the quantity. A pandas.DataFrame is passed as the sole argument to *collapse*, which must return a modified dataframe.

Returns Each key converts a Quantity into a pyam.IamDataFrame.

Return type list of Key

See also:

message_ix.reporting.computations.as_pyam()

The IAMC data format includes columns named 'Model', 'Scenario', 'Region', 'Variable', 'Unit'; one of 'Year' or 'Time'; and 'value'.

Using *convert_pyam()*:

- 'Model' and 'Scenario' are populated from the attributes of the Scenario returned by the Reporter key `scenario`;
- 'Variable' contains the name(s) of the *quantities*;
- 'Unit' contains the units associated with the *quantities*; and
- 'Year' or 'Time' is created according to *year_time_dim*.

A callback function (*collapse*) can be supplied that modifies the data before it is converted to an `IamDataFrame`; for instance, to concatenate extra dimensions into the 'Variable' column. Other dimensions can simply be dropped (with *drop*). Dimensions that are not collapsed or dropped will appear as additional columns in the resulting `IamDataFrame`; this is valid, but non-standard IAMC data.

For example, here the values for the MESSAGEix `technology` and `mode` dimensions are appended to the 'Variable' column:

```
def m_t(df):
    """Callback for collapsing ACT columns."""
    # .pop() removes the named column from the returned row
    df['variable'] = 'Activity|' + df['t'] + '|' + df['m']
    return df

ACT = rep.full_key('ACT')
keys = rep.convert_pyam(ACT, 'ya', collapse=m_t, drop=['t', 'm'])
```

write(*key*, *path*)

Compute *key* and write its value to the file at *path*.

In addition to the formats handled by `ixmp.Reporter.write()`, this version will write `pyam.IamDataFrame` to CSV or Excel files using built-in methods.

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, index, sums])	Add *computation* to the Reporter under *key*.
add_file(path[, key])	Add exogenous quantities from *path*.
add_product(name, *quantities[, sums])	Add a computation that takes the product of *quantities*.
aggregate(qty, tag, dims_or_groups[, ...])	Add a computation that aggregates *qty*.
apply(generator, *keys)	Add computations from *generator* applied to *key*.
check_keys(*keys)	Check that *keys* are in the Reporter.
configure([path])	Configure the Reporter.
describe([key, quiet])	Return a string describing the computations that produce *key*.
disaggregate(qty, new_dim[, method, args])	Add a computation that disaggregates *qty* using *method*.
finalize(scenario)	Prepare the Reporter to act on *scenario*.
full_key(name_or_key)	Return the full-dimensionality key for *name_or_key*.
get([key])	Execute and return the result of the computation *key*.
keys()	
read_config(path)	Configure the Reporter with information from a YAML file at *path*.
set_filters(**filters)	Apply *filters* ex ante (before computations occur).
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*, *index=False*, *sums=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.
- **index** (*bool, optional*) – If True, *key* is added to the index as a full-resolution key, so it can be later retrieved with *full_key()*.
- **sums** (*bool, optional*) – If True, all partial sums of *key* are also added to the Reporter.

Raises KeyError – If *key* is already in the Reporter; any key referred to by *computation* does not exist; or `sums=True` and the key for one of the partial sums of *key* is already in the Reporter.

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*, *sums=False*)

Add a computation that aggregates *qty*.

Parameters

- **qty** (*Key* or str) – Key of the quantity to be aggregated.
- **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** (`xarray.DataArray`, optional) – Weights for weighted aggregation.
- **keep** (*bool, optional*) – Passed to `computations.aggregate`.
- **sums** (*bool, optional*) – Passed to *add()*.

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*, *quiet=False*)
Return a string describing the computations that produce *key*.

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

The string is also printed. If *quiet*, do not print to the console.

disaggregate(*qty*, *new_dim*, *method='shares'*, *args=[]*)
Add a computation that disaggregates *qty* using *method*.

Parameters

- **qty** (*hashable*) – Key of the quantity 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_or_key*)
Return the full-dimensionality key for *name_or_key*.

An ixmp variable 'foo' with dimensions (a, c, n, q, x) is available in the Reporter as `'foo:a-c-n-q-x'`. This `Key` can be retrieved with:

```
rep.full_key('foo')
rep.full_key('foo:c')
# etc.
```

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 are stored in the reporter at the `'filters'` key, and are passed to `ixmp.Scenario.par()` and similar methods. All quantity values read from the Scenario are filtered *before* any other computations take place.

Parameters filters (*mapping of str → (list of str or None)*) – Argument names are dimension names; values are lists of allowable labels along the respective dimension, *or* None to clear any existing filters for the dimension.

If no arguments are provided, *all* filters are cleared.

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*.

class ixmp.reporting.**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. Keys **refer** to quantities, using three components:

1. a string `name`,
2. zero or more ordered dimensions `dims`, and
3. an optional `tag`.

For example, an ixmp parameter with three dimensions can be initialized with:

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

Key allows a specific, explicit reference to various forms of "foo":

- in its full resolution, i.e. indexed by a, b, and c:

```
>>> k1 = Key('foo', ['a', 'b', 'c'])
>>> k1
<foo:a-b-c>
```

- in a partial sum over one dimension, e.g. summed across dimension c, with remaining dimensions a and b:

```
>>> k2 = k1.drop('c')
>>> k2
<foo:a-b>
```

- in a partial sum over multiple dimensions, etc.:

```
>>> k1.drop('a', 'c') == k2.drop('a') == 'foo:b'
True
```

- after it has been manipulated by different reporting computations, e.g.

```
>>> k3 = k1.add_tag('normalized')
>>> k3
<foo:a-b-c:normalized>
>>> k4 = k3.add_tag('rescaled')
>>> k4
<foo:a-b-c:normalized+rescaled>
```

Notes:

A Key has the same hash, and compares equal to its `str` representation. `repr(key)` prints the Key in angle brackets ('<>') to signify that it is a Key object.

```
>>> str(k1)
'foo:a-b-c'
>>> repr(k1)
'<foo:a-b-c>'
>>> hash(k1) == hash('foo:a-b-c')
True
```

Keys are **immutable**: the properties `name`, `dims`, and `tag` are *read-only*, and the methods `append()`, `drop()`, and `add_tag()` return *new* Key objects.

Keys may be generated concisely by defining a convenience method:

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

add_tag(*tag*)
Return a new Key with *tag* appended.

append(**dims*)
Return a new Key with additional dimensions *dims*.

dims
Dimensions of the quantity, `tuple` of `str`.

drop(**dims*)
Return a new Key with *dims* dropped.

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

Parameters

- **value** (*str or Key*) – Value to use to generate a new Key.
- **drop** (*list of str, optional*) – Existing dimensions of *value* to drop. See `drop()`.
- **append** (*list of str, optional.*) – New dimensions to append to the returned Key. See `append()`.
- **tag** (*str, optional*) – Tag for returned Key. If *value* has a tag, the two are joined using a '+' character. See `add_tag()`.

Returns

Return type *Key*

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

name
Name of the quantity, `str`.

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

Dimensions are ordered by their first appearance:

1. First, the dimensions of the first of the *keys*.
2. Next, any additional dimensions in the second of the *keys* that were not already added in step 1.
3. etc.

Parameters **new_name** (`str`) – Name for the new Key. The names of *keys* are discarded.

tag
Quantity tag, `str`.

Computations

message_ix.reporting.computations.**add**(*a*, *b*, *fill_value=0.0*)
Sum of *a* and *b*.

message_ix.reporting.computations.**as_pyam**(*scenario*, *quantities*, *year_time_dim*, *drop=[]*, *collapse=None*)
Return a `pyam.IamDataFrame` containing *quantities*.

Warnings are logged if the arguments result in additional, unhandled columns in the resulting data frame that are not part of the IAMC spec.

Raises `ValueError` – If the resulting data frame has duplicate values in the standard IAMC index columns. `pyam.IamDataFrame` cannot handle this data.

See also:

`message_ix.reporting.Reporter.convert_pyam()`

message_ix.reporting.computations.**broadcast_map**(*quantity*, *map*, *rename={}*)
Broadcast *quantity* using a *map*.

The *map* must be a 2-dimensional quantity, such as returned by *`map_as_qty()`*.

quantity is 'broadcast' by multiplying it with the 2-dimensional *map*, and then dropping the common dimension. The result has the second dimension of *map* instead of the first.

Parameters **rename** (`dict (str -> str), optional`) – Dimensions to rename on the result.

message_ix.reporting.computations.**concat**(**args*)
Concatenate *args*, which must be `pyam.IamDataFrame`.

message_ix.reporting.computations.**map_as_qty**(*set_df*)
Convert *set_df* to a Quantity.

For the MESSAGE sets named `cat_*` (see *Category types and mappings*) `ixmp.Scenario.set()` returns a `DataFrame` with two columns: the *category* set (S1) elements and the *category member* set (S2) elements.

This computation converts the DataFrame *set_df* into a quantity with two dimensions. At the coordinates (s_1, s_2), the value is 1 if s_2 is a mapped from s_1; otherwise 0.

See also:

`broadcast_map()`

message_ix.reporting.computations.**write_report**(*quantity*, *path*)
Write the report identified by *key* to the file at *path*.

If *quantity* is a `pyam.IamDataFrame` and *path* ends with '.csv' or '.xlsx', use `pyam` methods to write the file to CSV or Excel format, respectively. Otherwise, equivalent to *`ixmp.reporting.computations.write_report()`*.

Computations from ixmp

Elementary computations for reporting.

Unless otherwise specified, these methods accept and return `Quantity` objects for data arguments/return values.

Calculations:

`aggregate`(quantity, groups, keep)	Aggregate *quantity* by *groups*.
`disaggregate_shares`(quantity, shares)	Disaggregate *quantity* by *shares*.
`product`(*quantities)	Return the product of any number of *quantities*.
`ratio`(numerator, denominator)	Return the ratio *numerator* / *denominator*.
`sum`(quantity[, weights, dimensions])	Sum *quantity* over *dimensions*, with optional *weights*.

Input and output:

`load_file`(path)	Read the file at *path* and return its contents.
`write_report`(quantity, path)	Write a quantity to a file.

Data manipulation:

`concat`(*objs, **kwargs)	Concatenate Quantity *objs*.

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

Aggregate *quantity* by *groups*.

Parameters

- **quantity** (`Quantity`) –
- **groups** (*dict of dict*) – Top-level keys are the names of dimensions in *quantity*. Second-level keys are group names; second-level values are lists of labels along the dimension to sum into a group.
- **keep** (`bool`) – If True, the members that are aggregated into a group are returned with the group sums. If False, they are discarded.

Returns Same dimensionality as *quantity*.

Return type `Quantity`

`ixmp.reporting.computations.`**`concat`**`(`**objs*, ***kwargs*`)`

Concatenate Quantity *objs*.

Any strings included amongst *args* are discarded, with a logged warning; these usually indicate that a quantity is referenced which is not in the Reporter.

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

Disaggregate *quantity* by *shares*.

`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.`**`product`**`(`**quantities*`)`

Return the product of any number of *quantities*.

`ixmp.reporting.computations.`**`ratio`**`(`*numerator*, *denominator*`)`

Return the ratio *numerator* / *denominator*.

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

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

ixmp.reporting.computations.**write_report**(*quantity*, *path*)
Write a quantity to a file.

Parameters **path** (*str or Path*) – Path to the file to be written.

Configuration

ixmp.reporting.configure([path])	Configure reporting globally.
ixmp.reporting.utils.RENAME_DIMS	Dimensions to rename when extracting raw data from Scenario objects.
ixmp.reporting.utils.REPLACE_UNITS	Replacements to apply to quantity units before parsing by pint.
ixmp.reporting.utils.UNITS	pint unit registry for processing quantity units.

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

Modifies global variables that affect the behaviour of *all* Reporters and computations, namely *RENAME_DIMS*, *REPLACE_UNITS*, and *UNITS*.

Valid configuration keys—passed as *config* keyword arguments—include:

Other Parameters

- **units** (*mapping*) – Configuration for handling of units. Valid sub-keys include:
  - **replace** (mapping of str -> str): replace units before they are parsed by pint. Added to *REPLACE_UNITS*.
  - **define** (str): block of unit definitions, added to *UNITS* so that units are recognized by pint. See the pint documentation.
- **rename_dims** (*mapping of str -> str*) – Update *RENAME_DIMS*.

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

message_ix.reporting.**PRODUCTS = (('out', ('output', 'ACT')), ('in', ('input', 'ACT')), ('re**
Automatic quantities that are the product() of two others.

message_ix.reporting.**DERIVED = [('tom:nl-t-yv-ya', (<function add>, 'fom:nl-t-yv-ya', 'vom**
Automatic quantities derived by other calculations.

message_ix.reporting.**PYAM_CONVERT = [('out:nl-t-ya-m-nd-c-l', 'ya', {'kind': 'ene', 'var'**
Quantities to automatically convert to IAMC format using *as_pyam()*.

message_ix.reporting.**REPORTS = {'message:costs': ['inv:pyam', 'fom:pyam', 'vom:pyam', 'tor**
Automatic reports that *concat()* quantities converted to IAMC format.

message_ix.reporting.**MAPPING_SETS = ['addon', 'emission', 'tec', 'year']**
MESSAGE mapping sets, converted to quantities using *map_as_qty()*.

ixmp.reporting.utils.**RENAME_DIMS = {'commodity': 'c', 'emission': 'e', 'grade': 'g', 'la**
Dimensions to rename when extracting raw data from Scenario objects. Mapping from Scenario dimension name -> preferred dimension name. message_ix adds the standard short symbols for MESSAGE sets to this variable.

ixmp.reporting.utils.**REPLACE_UNITS = {'%': 'percent'}**
Replacements to apply to quantity units before parsing by pint. Mapping from original unit -> preferred unit.

ixmp.reporting.utils.**UNITS = <pint.registry.UnitRegistry object>**

pint unit registry for processing quantity units. All units handled by `ixmp.reporting` must be either standard SI units, or added to this registry.

Utilities

class ixmp.reporting.attrseries.**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 available 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.**as_attrseries**(*obj*)

Convert obj to an AttrSeries.

ixmp.reporting.utils.**as_quantity**(*obj*)

Convert obj to an AttrSeries.

ixmp.reporting.utils.**as_sparse_xarray**(*obj*)

Convert *obj* to an `xarray.DataArray` with sparse.COO storage.

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.**dims_for_qty**(*data*)

Return the list of dimensions for *data*.

If *data* is a `pandas.DataFrame`, its columns are processed; otherwise it must be a list.

ixmp.reporting.RENAME_DIMS is used to rename dimensions.

`ixmp.reporting.utils.`**`filter_concat_args`**(*args*)
Filter out str and Key from *args*.

A warning is logged for each element removed.

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

`message_ix.reporting.pyam.`**`collapse_message_cols`**(*df*, *var*, *kind=None*)
`as_pyam()` *collapse*=... callback for MESSAGE quantities.

Parameters

- **var** (*str*) – Name for 'variable' column.
- **kind** (*None or 'ene' or 'emi', optional*) – Determines which other columns are combined into the 'region' and 'variable' columns:
  - 'ene': 'variable' is `'<var>|<level>|<commodity>|<technology>|<mode>'` and 'region' is `'<region>|<node_dest>'` (if *var='out'*) or `'<region>|<node_origin>'` (if *'var='in'*).
  - 'emi': 'variable' is `'<var>|<emission>|<technology>|<mode>'`.
  - Otherwise: 'variable' is `'<var>|<technology>'`.

  The referenced columns are also dropped, so it is not necessary to provide the *drop* argument of `as_pyam()`.

3.4.4 Model-building tools

Add model years to an existing Scenario

Description

This tool adds new modeling years to an existing *message_ix.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 after importing the add_year function:

```
from message_ix.tools.add_year import add_year
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:

```
$ message-ix \
    --platform default
    --model MESSAGE_Model \
    --scenario baseline \
    add-years
    --years_new 2015,2025,2035,2045
```

   For the full list of input arguments, run:

```
$ message-ix add-years --help
```

2. By calling the function *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*, *reg_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 based on one data point

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.

CHAPTER 4

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 What's New

4.1.1 v2.0.0 (2020-01-14)

message_ix v2.0.0 coincides with ixmp v2.0.0.

Migration notes

Support for **Python 2.7 is dropped** as it has reached end-of-life, meaning no further releases will be made even to fix bugs. See PEP-0373 and https://python3statement.org. `message_ix` users must upgrade to Python 3.

Command-line interface (CLI). Use `message-ix` as the program for all command-line operations:

- `message-ix copy-model` replaces `messageix-config`.
- `message-ix dl` replaces `messageix-dl`.
- `message-ix` also provides all the features of the `ixmp` CLI.

Configuration. ixmp adds a streamlined system for storing information about different platforms, backends, and databases that store Scenario data. See the ixmp release notes for migration notes.

All changes

- #285: Drop support for Python 2.

- #284: Add a suggested sequence/structure to how to run the Westeros tutorials.
- #281: Test and improve logic of `years_active()` and `vintage_and_active_years()`.
- #269: Enforce `year`-indexed columns as integers.
- #256: Update to use `ixmp.config` and improve CLI.
- #255: Add `message_ix.testing.nightly` and `message-ix nightly` CLI command group for slow-running tests.
- #249, #259: Build MESSAGE and MESSAGE_MACRO classes on ixmp model API; adjust Scenario.
- #235: Add a reporting tutorial.
- #236, #242, #263: Enhance reporting.
- #232: Add Westeros tutorial for modelling seasonality, update existing tutorials.
- #276: Improve add_year for bounds and code cleanup

4.1.2 v1.2.0 (2019-06-25)

MESSAGEix 1.2.0 adds an option to set the commodity balance to strict equality, rather than a supply > demand inequality. It also improves the support for models with non-equidistant years.

Other improvements include an experimental reporting module, support for CPLEX solver options via `solve()`, and a reusable `message_ix.testing` module.

Release 1.2.0 coincides with ixmp release 0.2.0, which provides full support for `clone()` across platforms (database instances), e.g. from a remote database to a local HSQL database; as well as other improvements. See the ixmp release notes for further details.

All changes

- #161: A feature for adding new periods to a scenario.
- #205: Implement required changes related to timeseries-support and cloning across platforms (see ixmp#142).
- #196: Improve testing by re-using `ixmp` apparatus.
- #187: Test for cumulative bound on emissions.
- #182: Fix cross-platform cloning.
- #178: Bugfix of the `PRICE_EMISSION` variable in models with non-equidistant period durations.
- #176: Add `message_ix.reporting` module.
- #173: The meth:~.*Scenario.solve* command now takes additional arguments when solving with CPLEX. The cplex.opt file is now generated on the fly during the solve command and removed after successfully solving.
- #172: Add option to set `COMMODITY_BALANCE` to equality.
- #154: Enable documentation build on ReadTheDocs.
- #138: Update documentation and tutorials.
- #131: Update clone function argument *scen* to *scenario* with planned deprecation of the former.

4.1.3 v1.1.0 (2018-11-21)

Migration notes

This patch introduces a few backwards-incompatible changes to database management.

Database Migration

If you see an error message like:

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
usr/local/lib/python2.7/site-packages/ixmp/core.py:81: in __init__
    self._jobj = java.ixmp.Platform("Python", dbprops)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <jpype._jclass.at.ac.iiasa.ixmp.Platform object at 0x7ff1a8e98410>
args = ('Python', '/tmp/kH07wz/test.properties')

    def _javaInit(self, *args):
        object.__init__(self)

        if len(args) == 1 and isinstance(args[0], tuple) \
           and args[0][0] is _SPECIAL_CONSTRUCTOR_KEY:
            self.__javaobject__ = args[0][1]
        else:
            self.__javaobject__ = self.__class__.__javaclass__.newClassInstance(
>               *args)
E           org.flywaydb.core.api.FlywayExceptionPyRaisable: org.flywaydb.core.api.
↪FlywayException: Validate failed: Migration checksum mismatch for migration 1
E           -> Applied to database : 1588531206
E           -> Resolved locally    : 822227094
```

Then you need to update your local database. There are two methods to do so:

1. Delete it (you will lose all data and need to regenerate it). The default location is ~/.local/ixmp/localdb/.
2. Manually apply the underlying migrations. This is not particularly easy, but allows you to save all your data. If you want help, feel free to get in contact on the listserv.

New Property File Layout

If you see an error message like:

```
usr/local/lib/python2.7/site-packages/jpype/_jclass.py:111: at.ac.iiasa.ixmp.
↪exceptions.IxExceptionPyRaisable
---------------------------- Captured stdout setup -----------------------------
2018-11-13 08:15:17,410 ERROR at.ac.iiasa.ixmp.database.DbConfig:357 - missing␣
↪property 'config.server.config' in /tmp/hhvE1o/test.properties
2018-11-13 08:15:17,412 ERROR at.ac.iiasa.ixmp.database.DbConfig:357 - missing␣
↪property 'config.server.password' in /tmp/hhvE1o/test.properties
2018-11-13 08:15:17,412 ERROR at.ac.iiasa.ixmp.database.DbConfig:357 - missing␣
↪property 'config.server.username' in /tmp/hhvE1o/test.properties
2018-11-13 08:15:17,413 ERROR at.ac.iiasa.ixmp.database.DbConfig:357 - missing␣
↪property 'config.server.url' in /tmp/hhvE1o/test.properties
------------------------------ Captured log setup ------------------------------
```

(continues on next page)

(continued from previous page)

```
core.py                         80 INFO     launching ixmp.Platform using config file at
↪'/tmp/hhvE1o/test.properties'
_________________ ERROR at setup of test_add_spatial_multiple __________________

    @pytest.fixture(scope="session")
    def test_mp():
        test_props = create_local_testdb()

        # start jvm
        ixmp.start_jvm()

        # launch Platform and connect to testdb (reconnect if closed)
>       mp = ixmp.Platform(test_props)
```

Then you need to update your property configuration file. The old file looks like:

```
config.name = message_ix_test_db@local
jdbc.driver.1 = org.hsqldb.jdbcDriver
jdbc.url.1 = jdbc:hsqldb:file:/path/to/database
jdbc.user.1 = ixmp
jdbc.pwd.1 = ixmp
jdbc.driver.2 = org.hsqldb.jdbcDriver
jdbc.url.2 = jdbc:hsqldb:file:/path/to/database
jdbc.user.2 = ixmp
jdbc.pwd.2 = ixmp
```

The new file should look like:

```
config.name = message_ix_test_db@local
jdbc.driver = org.hsqldb.jdbcDriver
jdbc.url = jdbc:hsqldb:file:/path/to/database
jdbc.user = ixmp
jdbc.pwd = ixmp
```

All changes

- #202: Added the "Development rule of thumb" section from the wiki and the Tutorial style guide to the Contributor guidelines. Tweaked some formatting to improve readibility.
- #113: Upgrading to MESSAGEix 1.1: improved representation of renewables, share constraints, etc.
- #109: MACRO module added for initializing models to be solved with MACRO. Added scenario-based CI on circleci.
- #99: Fixing an error in the compuation of the auxiliary GAMS reporting variable `PRICE_EMISSION`.
- #89: Fully implementing system reliability and flexibity considerations (cf. Sullivan).
- #88: Reformulated capacity maintainance constraint to ensure that newly installed capacity cannot be decommissioned within the same model period as it is built in.
- #84: `message_ix.Scenario.vintage_active_years()` now limits active years to those after the first model year or the years of a certain technology vintage.
- #82: Introducing "add-on technologies" for mitigation options, etc.
- #81: Share constraints by mode added.
- #80: Share constraints by commodity/level added.

- #78: Bugfix: `message_ix.Scenario.solve()` uses 'MESSAGE' by default, but can be provided other model names.
- #77: `rename()` function can optionally keep old values in the model (i.e., copy vs. copy-with-replace).
- #74: Activity upper and lower bounds can now be applied to all modes of a technology.
- #67: Use of advanced basis in cplex.opt turned off by default to avoid conflicts with barrier method.
- #65: Bugfix for downloading tutorials. Now downloads current installed version by default.
- #60: Add basic ability to write and read model input to/from Excel.
- #59: Added MacOSX CI support.

4.2 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.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 4. Give notice of publication

E-mail <message_ix@iiasa.ac.at> with notice of notice of any new or pending publication.

4.2.5 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

4.3 Contributing to MESSAGE*ix* development

We welcome contributions to the code base and development of new features for the MESSAGE*ix* framework. This page contains guidelines for making these contributions.

- *Filing issues for bugs and enhancements*
- *Contributing code via Github PRs*
  - *1. Choose a repository*
  - *2. Fork, branch, and open a pull request*
  - *3. Ensure checks pass*
  - *4. Review*
  - *Other tips*
- *Code style*
- *Versions and releases*
- *Contributing tutorials*
  - *Coding & writing style*
  - *Structure*
  - *Location*

4.3.1 Filing issues for bugs and enhancements

We use Github **issues** for several purposes:

- Ask and answer *questions* about intended behaviour or issues running the framework or related models.
- Report *bugs*, i.e. unintended or undocumented behaviour.
- Request *changes* to exiting behaviour.
- Request specific *enhancements* and *new features*, both urgent and long-term/low-priority.
- Discuss and design of other improvements.

Please search through open *and* closed issues for *both* the message_ix and ixmp repositories. Review any related issues. Then, if your issue is not found, open a new one.

4.3.2 Contributing code via Github PRs

See the short introduction to the Github flow, which describes a **pull request** and how it is used. Use online documentation for git, Github, and Python to ensure you are able to complete the process below. Register a Github account, if you do not already have one.

1. Choose a repository

Decide which part of the MESSAGE*ix* software stack is the appropriate location for your code:

ixmp Contributions not specific to MESSAGE*ix* model framework, e.g. that could be used for other, non-MESSAGE models.

> **ixmp_source** Java / JDBC backend for `ixmp`.

message_ix Contributions not specific to *any particular MESSAGEix* model instance. Additions to `message_ix` should be usable in any MESSAGE-scheme model.

message_data or message_doc Contributions to the MESSAGE-GLOBIOM family of models, including the global model; and its documentation, respectively.

2. Fork, branch, and open a pull request

Fork the chosen repository to your own Github account. Create a branch with an appropriate name:

- `all-lower-case-with-hyphens`.
- `issue/1234` if you are addressing a specific issue.
- `feature/do-something` if you are adding a new feature.

Open a PR (e.g. on message_ix) to merge your code into the `master` branch. The `message_ix` and `ixmp` repositories each have a template for the text of the PR, including the minimum requirements:

- A title and one-sentence summary of the change. This is like the abstract of a publication: it should help a developer/reviewer/user quickly learn what the PR is about.
- Confirm that unit or integration tests have been added or revised to cover the changed code, and that the tests pass (see below).
- Confirm that documentation of the API and its usage is added or revised as necessary.
- Add a line to the file `RELEASE_NOTES.md` describing the changes (use the same title or one-sentence summary as above) and linking to the PR.

Optionally:

- Include a longer description of the design, or any changes whose purpose is not clear by inspecting code.
- Put “WIP:” or the construction sign Unicode character () at the start of the PR title to indicate “work in progress” while you continue to add commits; or use GitHub’s ‘draft’ pull requests. This is good development practice: it ensures the automatic checks pass as you add to the code on your branch.

3. Ensure checks pass

MESSAGE*ix* currently has three kinds of **automatic**, or “continuous integration” checks:

- The CLA Assistant ensures you have signed the *Contributor License Agreement* (text below). All contributors are required to sign the CLA 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 Stickler service reviews Python code style (see below).
- Travis (Linux, macOS) and AppVeyor (Windows) run the test suite in `tests/`.

Resolve any non-passing checks—seeking help if needed.

If your PR updates the documentation, **manually** check that it can be built. See `doc/README.rst`.

4. Review

Using the GitHub sidebar on your PR, request a review from another MESSAGE*ix* contributor. GitHub suggests reviewers; optionally, contact the IIASA Energy Program to ask who should review your code. Address any comments raised by the reviewer, who will also merge your PR when it is ready.

Other tips

- If other PRs are merged before yours, a **merge conflict** may arise and must be addressed to complete the above steps. This means that your PR, and the other PR, both modify the same file(s) in the same location(s), and git cannot automatically determine which changes to use. Learn how to:
  - git merge. This brings all updates from the `master` branch into your PR branch, giving you a chance to fix conflicts and make a new commit.
  - git rebase. This replays your PR branch commits one-by-one, starting from the tip of the `master` branch (rather than the original starting commit).

4.3.3 Code style

- Python: follow PEP 8.
- R: follow the style of the existing code base.
- Jupyter notebooks (`.ipynb`): see below, under *Contributing tutorials*.
- 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.

4.3.4 Versions and releases

- We use semantic versioning.
- We keep at least two active milestones on each of the message_ix and ixmp repositories:
  - The next minor version. E.g. if the latest release was 3.5, the next minor release/milestone is 3.6.
  - The next major version. E.g. 4.0.
- The milestones are closed at the time a new version is released. If a major release (e.g. 4.0) is made without the preceding minor release (e.g. 3.6), both are closed together.
- Every issue and PR must be assigned to a milestone to record the decision/intent to release it at a certain time.
- New releases are made by Energy Program staff using the Release procedure, and appear on Github, PyPI, and conda-forge.

4.3.5 Contributing tutorials

Developers *and users* of the MESSAGE*ix* framework are welcome to contribute **tutorials**, according to the following guidelines. Per the license and CLA, tutorials will become part of the message_ix test suite and will be publicly available.

Developers **must** ensure new features (including `message_ix.tools` submodules) are fully documented. This can be done via the API documentation (this site) and, optionally, a tutorial. These have complementary purposes:

- The API documentation (built using Sphinx and ReadTheDocs) must completely, but succinctly, *describe the arguments and behaviour* of every class and method in the code.
- Tutorials serve as *structured learning exercises* for the classroom or self-study. The intended learning outcome for each tutorial is that students understand how the model framework API may be used for scientific research, and can begin to implement their own models or model changes.

Coding & writing style

- Tutorials are formatted as Jupyter notebooks in Python or R.
- Commit 'bare' notebooks in git, i.e. without cell output. Notebooks will be run and rendered when the documentation is generated.
- Add a line to `tests/test_tutorials.py`, so that the parametrized test function runs the tutorial (as noted at #196).
- Optionally, use Jupyter notebook slide-show features so that the tutorial can be viewed as a presentation.
- When relevant, provide links to publications or sources that provide greater detail for the methodology, data, or other packages used.
- Providing the mathematical formulation in the tutorial itself is optional.
- Framework specific variables and parameters or functions must be in italic.
- Relevant figures, tables, or diagrams should be added to the tutorial if these can help users to understand concepts.
  - Place rendered versions of graphics in a directory with the tutorial (see *Location* below). Use SVG, PNG, JPG, or other web-ready formats.

Structure

Generally, a tutorial should have the following elements or sections.

- Tutorial introduction:
  - The general overview of tutorial.
  - The intended learning outcome.
  - An explanation of which features are covered.
  - Reference and provide links to any tutorials that are interlinked or part of a series.
- Description of individual steps:
  - A brief explanation of the step.
  - A link to any relevant mathematical documentation.
- Modeling results and visualizations:
  - Model outputs and post-processing calculations in tutorials should demonstrate usage of the *message_ix.reporting module*.
  - Plots to depict results should use pyam.
  - Include a brief discussion of insights from the results, in line with the learning objectives.
- Exercises: include self-test questions, small activities, and exercises at the end of a tutorial so that users (and instructors, if any) can check their learning.

Location

Place notebooks in an appropriate location:

`tutorial/name.ipynb`: Stand-alone tutorial.

`tutorial/example/example_baseline.ipynb`: Group of tutorials named "example." Each notebook's file name begins with the group name, followed by a name beginning with underscores. The group name can refer to a specific RES shared across multiple tutorials. Some example names include:

```
<group>_baseline.ipynb

<group>_basic.ipynb  # Basic modeling features, e.g.:
<group>_emmission_bounds.ipynb
<group>_emission_taxes.ipynb
<group>_fossil_resources.ipynb

<group>_adv.ipynb  # Advanced modeling features, e.g.:
<group>_addon_technologies.ipynb
<group>_share_constraints.ipynb

<group>_renewables.ipynb  # Features related to renewable energy, e.g.:
<group>_firm_capacity.ipynb
<group>_flexible_generation.ipynb
<group>_renewable_resources.ipynb
```

4.4 Contributor License Agreement

4.4.1 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.

4.4.2 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.

4.4.3 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.

4.4.4 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.

4.4.5 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.

4.4.6 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.

4.5 Frequently asked questions

4.5.1 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.

4.5.2 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.

4.5.3 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.

4.6 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

ixmp.reporting.computations, 70
ixmp.reporting.utils, 73

m

message_ix.models, 23
message_ix.reporting.computations, 70
message_ix.reporting.pyam, 74
message_ix.testing, 24
message_ix.tools.add_year, 76
message_ix.utils, 24

Index

A

add() (*in module message_ix.reporting.computations*), 70
add() (*ixmp.reporting.Reporter method*), 64
add_cat() (*message_ix.Scenario method*), 13
add_file() (*ixmp.reporting.Reporter method*), 65
add_geodata() (*message_ix.Scenario method*), 13
add_horizon() (*message_ix.Scenario method*), 14
add_par() (*message_ix.Scenario method*), 14
add_product() (*ixmp.reporting.Reporter method*), 65
add_set() (*message_ix.Scenario method*), 14
add_spatial_sets() (*message_ix.Scenario method*), 14
add_tag() (*ixmp.reporting.Key method*), 69
add_timeseries() (*message_ix.Scenario method*), 15
add_year() (*in module message_ix.tools.add_year*), 76
add_year_par() (*in module message_ix.tools.add_year*), 76
add_year_set() (*in module message_ix.tools.add_year*), 77
aggregate() (*in module ixmp.reporting.computations*), 71
aggregate() (*ixmp.reporting.Reporter method*), 65
append() (*ixmp.reporting.Key method*), 69
apply() (*ixmp.reporting.Reporter method*), 65
as_attrseries() (*in module ixmp.reporting.utils*), 73
as_pyam() (*in module message_ix.reporting.computations*), 70
as_quantity() (*in module ixmp.reporting.utils*), 73
as_sparse_xarray() (*in module ixmp.reporting.utils*), 73
AttrSeries (*class in ixmp.reporting.attrseries*), 73

B

broadcast_map() (*in module message_ix.reporting.computations*), 70

C

cat() (*message_ix.Scenario method*), 15
cat_list() (*message_ix.Scenario method*), 15
change_scalar() (*message_ix.Scenario method*), 15
check_keys() (*ixmp.reporting.Reporter method*), 66
check_out() (*message_ix.Scenario method*), 15
clean_units() (*in module ixmp.reporting.utils*), 73
clone() (*message_ix.Scenario method*), 16
collapse_message_cols() (*in module message_ix.reporting.pyam*), 74
collect_units() (*in module ixmp.reporting.utils*), 73
commit() (*message_ix.Scenario method*), 16
concat() (*in module ixmp.reporting.computations*), 71
concat() (*in module message_ix.reporting.computations*), 70
configure() (*ixmp.reporting method*), 72
configure() (*ixmp.reporting.Reporter method*), 66
convert_pyam() (*message_ix.reporting.Reporter method*), 62

D

data_for_quantity() (*in module ixmp.reporting.utils*), 73
DEFAULT_CPLEX_OPTIONS (*in module message_ix.models*), 23
default_key (*ixmp.reporting.Reporter attribute*), 66
defaults (*message_ix.models.MESSAGE attribute*), 23
DERIVED (*in module message_ix.reporting*), 72
describe() (*ixmp.reporting.Reporter method*), 66
dims (*ixmp.reporting.Key attribute*), 69
dims_for_qty() (*in module ixmp.reporting.utils*), 73
disaggregate() (*ixmp.reporting.Reporter method*), 66
disaggregate_shares() (*in module ixmp.reporting.computations*), 71

discard_changes() (message_ix.Scenario method), 16
drop() (ixmp.reporting.Key method), 69

E

equ() (message_ix.Scenario method), 16
equ_list() (message_ix.Scenario method), 16

F

filter_concat_args() (in module ixmp.reporting.utils), 73
finalize() (ixmp.reporting.Reporter method), 66
firstmodelyear (message_ix.Scenario attribute), 16
from_scenario() (ixmp.reporting.Reporter class method), 66
from_str_or_key() (ixmp.reporting.Key class method), 69
from_url() (message_ix.Scenario class method), 16
full_key() (ixmp.reporting.Reporter method), 67

G

GAMS_min_version (message_ix.models.MESSAGE_MACRO attribute), 23
get() (ixmp.reporting.Reporter method), 67
get_geodata() (message_ix.Scenario method), 17
get_meta() (message_ix.Scenario method), 17
graph (ixmp.reporting.Reporter attribute), 64

H

has_equ() (message_ix.Scenario method), 17
has_par() (message_ix.Scenario method), 17
has_set() (message_ix.Scenario method), 17
has_solution() (message_ix.Scenario method), 17
has_var() (message_ix.Scenario method), 17

I

idx_names() (message_ix.Scenario method), 17
idx_sets() (message_ix.Scenario method), 17
init_equ() (message_ix.Scenario method), 17
init_par() (message_ix.Scenario method), 18
init_scalar() (message_ix.Scenario method), 18
init_set() (message_ix.Scenario method), 18
init_var() (message_ix.Scenario method), 18
interpolate_1d() (in module message_ix.tools.add_year), 77
interpolate_2d() (in module message_ix.tools.add_year), 77
intpol() (in module message_ix.tools.add_year), 78
is_default() (message_ix.Scenario method), 18
iter_sums() (ixmp.reporting.Key method), 69
ixmp.reporting.computations (module), 70
ixmp.reporting.utils (module), 73

K

Key (class in ixmp.reporting), 68
keys_for_quantity() (in module ixmp.reporting.utils), 74

L

last_update() (message_ix.Scenario method), 18
load_file() (in module ixmp.reporting.computations), 71
load_scenario_data() (message_ix.Scenario method), 18

M

make_dantzig() (in module message_ix.testing), 24
make_df() (in module message_ix.utils), 24
make_westeros() (in module message_ix.testing), 24
map_as_qty() (in module message_ix.reporting.computations), 70
MAPPING_SETS (in module message_ix.reporting), 72
mask_df() (in module message_ix.tools.add_year), 78
MESSAGE (class in message_ix.models), 23
message_ix.models (module), 23
message_ix.reporting.computations (module), 70
message_ix.reporting.pyam (module), 74
message_ix.testing (module), 24
message_ix.tools.add_year (module), 76
message_ix.utils (module), 24
MESSAGE_MACRO (class in message_ix.models), 23

N

name (ixmp.reporting.Key attribute), 69
name (message_ix.models.MESSAGE attribute), 23
name (message_ix.models.MESSAGE_MACRO attribute), 23

P

par() (message_ix.Scenario method), 19
par_list() (message_ix.Scenario method), 19
preload_timeseries() (message_ix.Scenario method), 19
product() (in module ixmp.reporting.computations), 71
product() (ixmp.reporting.Key class method), 69
PRODUCTS (in module message_ix.reporting), 72
PYAM_CONVERT (in module message_ix.reporting), 72

R

ratio() (in module ixmp.reporting.computations), 71
read_config() (ixmp.reporting.Reporter method), 67
read_excel() (message_ix.Scenario method), 19

read_version() (*message_ix.models.MESSAGE class method*), 23
remove_geodata() (*message_ix.Scenario method*), 19
remove_par() (*message_ix.Scenario method*), 19
remove_set() (*message_ix.Scenario method*), 19
remove_solution() (*message_ix.Scenario method*), 20
remove_timeseries() (*message_ix.Scenario method*), 20
rename() (*message_ix.Scenario method*), 20
RENAME_DIMS (*in module ixmp.reporting.utils*), 72
REPLACE_UNITS (*in module ixmp.reporting.utils*), 72
Reporter (*class in ixmp.reporting*), 63
Reporter (*class in message_ix.reporting*), 62
REPORTS (*in module message_ix.reporting*), 72
run() (*message_ix.models.MESSAGE method*), 23
run_id() (*message_ix.Scenario method*), 20

S

scalar() (*message_ix.Scenario method*), 20
Scenario (*class in message_ix*), 13
set() (*message_ix.Scenario method*), 20
set_as_default() (*message_ix.Scenario method*), 21
set_filters() (*ixmp.reporting.Reporter method*), 67
set_list() (*message_ix.Scenario method*), 21
set_meta() (*message_ix.Scenario method*), 21
slice_df() (*in module message_ix.tools.add_year*), 78
solve() (*message_ix.Scenario method*), 21
sum() (*in module ixmp.reporting.computations*), 71

T

tag (*ixmp.reporting.Key attribute*), 69
timeseries() (*message_ix.Scenario method*), 21
to_excel() (*message_ix.Scenario method*), 21

U

unit_uniform() (*in module message_ix.tools.add_year*), 78
UNITS (*in module ixmp.reporting.utils*), 72

V

var() (*message_ix.Scenario method*), 22
var_list() (*message_ix.Scenario method*), 22
vintage_and_active_years() (*message_ix.Scenario method*), 22
visualize() (*ixmp.reporting.Reporter method*), 67

W

write() (*ixmp.reporting.Reporter method*), 67
write() (*message_ix.reporting.Reporter method*), 63
write_report() (*in module ixmp.reporting.computations*), 71
write_report() (*in module message_ix.reporting.computations*), 70

Y

years_active() (*message_ix.Scenario method*), 22