Silicone Documentation

Release 1.3.0

Robin Lamboll, Zebedee Nicholls, Jarmo Kikstra

Oct 19, 2022

DOCUMENTATION

Silicone is a Python package which can be used to infer emissions from other emissions data. It is intended to 'infill' integrated assessment model (IAM) data so that their scenarios quantify more climate-relevant emissions than are natively reported by the IAMs themselves. It does this by comparing the incomplete emissions set to complete data from other sources. It uses the relationships within the complete data to make informed infilling estimates of otherwise missing emissions timeseries. For example, it can add emissions of aerosol precurors based on carbon dioxide emissions and infill nitrous oxide emissions based on methane, or split HFC emissions pathways into emissions of different specific HFC gases.

Silicone is free software under a BSD 3-Clause License, see LICENSE.

CHAPTER

ONE

INSTALLATION

Silicone can be installed with pip

```
pip install silicone
```

If you also want to run the example notebooks, install additional dependencies using

```
pip install silicone[notebooks]
```

Coming soon Silicone can also be installed with conda

```
conda install -c conda-forge silicone
```

CHAPTER

TWO

USAGE

All of our usage examples are included in `silicone/notebooks`.

CHAPTER

THREE

DEVELOPMENT

If you're interested in contributing to Silicone, we'd love to have you on board! This section of the docs will (once we've written it) detail how to get setup to contribute and how best to communicate.

3.1 Contributing

All contributions are welcome, some possible suggestions include:

- tutorials (or support questions which, once solved, result in a new tutorial :D)
- blog posts
- improving the documentation
- bug reports
- feature requests
- pull requests

Please report issues or discuss feature requests in the Silicone issue tracker. If your issue is a feature request or a bug, please use the templates available, otherwise, simply open a normal issue :)

As a contributor, please follow a couple of conventions:

- Create issues in the Silicone issue tracker for changes and enhancements, this ensures that everyone in the community has a chance to comment
- Be welcoming to newcomers and encourage diverse new contributors from all backgrounds: see the Python Community Code of Conduct
- Only push to your own branches, this allows people to force push to their own branches as they need without fear or causing others headaches
- Start all pull requests as draft pull requests and only mark them as ready for review once they've been rebased onto master, this makes it much simpler for reviewers
- Try and make lots of small pull requests, this makes it easier for reviewers and faster for everyone as review time grows exponentially with the number of lines in a pull request

3.2 Getting setup

To get setup as a developer, we recommend the following steps (if any of these tools are unfamiliar, please see the resources we recommend in *Development tools*):

1. Install conda and make
2. Run `make virtual-environment`, if that fails you can try doing it manually
    1. Change your current directory to Silicone's root directory (i.e. the one which contains `README.rst`), `cd silicone`
    2. Create a virtual environment to use with Silicone `python3 -m venv venv`
    3. Activate your virtual environment `source ./venv/bin/activate`
    4. Upgrade pip `pip intall --upgrade pip`
    5. Install the development dependencies (very important, make sure your virtual environment is active before doing this) `pip install -e .[dev]`
3. Make sure the tests pass by running `make test-all`, if that fails the commands are
    1. Activate your virtual environment `source ./venv/bin/activate`
    2. Run the unit and integration tests `pytest --cov -r a --cov-report term-missing`
    3. Test the notebooks `pytest -r a --nbval ./notebooks --sanitize ./notebooks/tests_sanitize.cfg`

3.2.1 Getting help

Whilst developing, unexpected things can go wrong (that's why it's called 'developing', if we knew what we were doing, it would already be 'developed'). Normally, the fastest way to solve an issue is to contact us via the issue tracker. The other option is to debug yourself. For this purpose, we provide a list of the tools we use during our development as starting points for your search to find what has gone wrong.

Development tools

This list of development tools is what we rely on to develop Silicone reliably and reproducibly. It gives you a few starting points in case things do go inexplicably wrong and you want to work out why. We include links with each of these tools to starting points that we think are useful, in case you want to learn more.

- Git
- Make
- Conda virtual environments
- Pip and pip virtual environments
- Tests
  - we use a blend of pytest and the inbuilt Python testing capabilities for our tests so checkout what we've already done in `tests` to get a feel for how it works
- Continuous integration (CI)
  - we use Travis CI for our CI but there are a number of good providers
- Jupyter Notebooks
  - Jupyter is automatically included in your virtual environment if you follow our *Getting setup* instructions
- Sphinx

Other tools

We also use some other tools which aren't necessarily the most familiar. Here we provide a list of these along with useful resources.

- Regular expressions
  - we use regex101.com to help us write and check our regular expressions, make sure the language is set to Python to make your life easy!

3.3 Formatting

To help us focus on what the code does, not how it looks, we use a couple of automatic formatting tools. These automatically format the code for us and tell use where the errors are. To use them, after setting yourself up (see *Getting setup*), simply run `make format`. Note that `make format` can only be run if you have committed all your work i.e. your working directory is 'clean'. This restriction is made to ensure that you don't format code without being able to undo it, just in case something goes wrong.

3.4 Buiding the docs

After setting yourself up (see *Getting setup*), building the docs is as simple as running `make docs` (note, run `make -B docs` to force the docs to rebuild and ignore make when it says '... index.html is up to date'). This will build the docs for you. You can preview them by opening `docs/build/html/index.html` in a browser.

For documentation we use Sphinx. To get ourselves started with Sphinx, we started with this example then used Sphinx's getting started guide.

3.4.1 Gotchas

To get Sphinx to generate pdfs (rarely worth the hassle), you require Latexmk. On a Mac this can be installed with `sudo tlmgr install latexmk`. You will most likely also need to install some other packages (if you don't have the full distribution). You can check which package contains any missing files with `tlmgr search --global --file [filename]`. You can then install the packages with `sudo tlmgr install [package]`.

3.4.2 Docstring style

For our docstrings we use numpy style docstrings. For more information on these, here is the full guide and the quick reference we also use.

3.5 Releasing

3.5.1 First step

1. Test installation with dependencies `make test-install`
2. Update `CHANGELOG.rst`
    - add a header for the new version between `master` and the latest bullet point
    - this should leave the section underneath the master header empty
3. `git add .`
4. `git commit -m "Release vX.Y.Z"`
5. `git tag vX.Y.Z`
6. Test version updated as intended with `make test-install`

3.5.2 Hitting the button

Our releases are handled automatically as part of our CI-CD pipeline. Hence all that needs to be done now is simply push to the repository, this will trigger an automatic release to PyPI.

1. `git push`
2. `git push --tags`

If the pipeline fails, one of the developers will receive a notification (and the reasons can then be debugged). If the pipeline passes, it's worth going to Silicone's PyPI and checking that the new release is as intended. You can also check that a fresh install would install the released version using `make test-pypi-install`.

3.6 Why is there a `Makefile` in a pure Python repository?

Whilst it may not be standard practice, a `Makefile` is a simple way to automate general setup (environment setup in particular). Hence we have one here which basically acts as a notes file for how to do all those little jobs which we often forget e.g. setting up environments, running tests (and making sure we're in the right environment), building docs, setting up auxillary bits and pieces.

CHAPTER

FOUR

DATABASE CRUNCHERS API

Database crunchers.

The classes within this module can be used to crunch a database of scenarios. Each 'Cruncher' has methods which return functions which can then be used to infill emissions detail (i.e. calculate 'follower' timeseries) based on 'lead' emissions timeseries.

4.1 Closest RMS cruncher API

Module for the database cruncher which uses the 'closest RMS' technique.

class `silicone.database_crunchers.rms_closest.`**`RMSClosest`**(*db*)

Bases: `_DatabaseCruncher`

Database cruncher which uses the 'closest RMS' technkque.

This cruncher derives the relationship between two or more variables by finding the scenario which has the most similar timeseries for the lead gases in the database. The follower gas timeseries is then simply copied from the closest scenario.

Here, 'most similar' is defined as the smallest time-averaged root mean squared (L2) difference. If multiple lead values are used, they may be weighted differently to account for differences between the reported units. The most similar model/scenario combination minimises

$$RMSerror = \sum_l w_l \left(\frac{1}{n} \sum_{t=0}^{n} (E_l(t) - e_l(t))^2 \right)^{1/2}$$

where l is a lead gas, w_l is a weighting for that lead gas, n is the total number of timesteps in all lead gas timeseries, $E_l(t)$ is the lead gas emissions timeseries and $e_l(t)$ is a lead gas emissions timeseries in the infiller database.

`derive_relationship`(*variable_follower*, *variable_leaders*, *weighting=None*)

Derive the relationship between the lead and the follow variables from the database.

Parameters

- **`variable_follower`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|C5F12"`).
- **`variable_leaders`** (*`list[str]`*) – The variable we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`). This may contain multiple elements.

- **weighting** (*dict{str: float}*) – When used with multiple lead variables, this weighting factor controls the relative importance of different variables for determining closeness. E.g. if wanting to compare both CO2 and CH4 emissions reported in mass units but weighted by the AR5 GWP100 metric, this would be {"Emissions|CO2": 1, "Emissions|CH4": 28}.

Returns
Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type
`func`

Raises

- **ValueError** – `variable_leaders` contains more than one variable.
- **ValueError** – There is no data for `variable_leaders` or `variable_follower` in the database.

4.2 Constant ratio cruncher API

Module for the database cruncher which uses the 'constant given ratio' technique.

class `silicone.database_crunchers.constant_ratio.`**`ConstantRatio`**(*db=None*)

Bases: `_DatabaseCruncher`

Database cruncher which uses the 'constant given ratio' technique.

This cruncher does not require a database upon initialisation. Instead, it requires a constant and a unit to be input when deriving relations. This constant, s, is the ratio of the follower variable to the lead variable i.e.:

$$E_f(t) = s * E_l(t)$$

where $E_f(t)$ is emissions of the follower variable and $E_l(t)$ is emissions of the lead variable.

`derive_relationship`(*variable_follower*, *variable_leaders*, *ratio*, *units*)

Derive the relationship between two variables from the database.

Parameters

- **variable_follower** (*str*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|C5F12"`).
- **variable_leaders** (*list[str]*) – The variable we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **ratio** (*float*) – The ratio between the leader and the follower data
- **units** (*str*) – The units of the follower data.

Returns
Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two.

Return type
`func`

4.3 Equal quantile walk cruncher API

Module for the database cruncher which uses the 'equal quantile walk' technique.

class `silicone.database_crunchers.equal_quantile_walk.`**`EqualQuantileWalk`**(*db*)

Bases: `_DatabaseCruncher`

Database cruncher which uses the 'equal quantile walk' technique.

This cruncher assumes that the amount of effort going into reducing one emission set is equal to that for another emission, therefore the lead and follow data should be at the same quantile of all pathways in the infiller database. It calculates the quantile of the lead infillee data in the lead infiller database, then outputs that quantile of the follow data in the infiller database.

`derive_relationship`(*variable_follower*, *variable_leaders*, *smoothing=None*, *weighting=None*)

Derive the relationship between two variables from the database.

Parameters

- **`variable_follower`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|C5F12"`).
- **`variable_leaders`** (*`list[str]`*) – The variable we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **`smoothing`** (*`float or string`*) – By default, no smoothing is done on the distribution. If a value is provided, it is fed into `scipy.stats.gaussian_kde()` - see full documentation there. In short, if a float is input, we fit a Gaussian kernel density estimator with that width to the points. If a string is used, it must be either "scott" or "silverman", after those two methods of determining the best kernel bandwidth.
- **`weighting`** (*`Dict{(str, str) : float}`*) – The dictionary, mapping the (mode, scenario) tuple onto the weight (relative to a weight of 1 for the default). This does not have to include all scenarios in df, but cannot include scenarios not in df.

Returns

Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type

`func`

Raises

- **`ValueError`** – `variable_leaders` contains more than one variable.
- **`ValueError`** – There is no data for `variable_leaders` or `variable_follower` in the database.

4.4 Interpolate specified scenarios and models cruncher API

`class silicone.database_crunchers.interpolate_specified_scenarios_and_models.ScenarioAndModelSpecificIn`

Bases: `_DatabaseCruncher`

Database cruncher which pre-filters to only use data from specific scenarios, then runs the interpolation cruncher to return values from that set of scenarios. See the documentation of Interpolation for more details.

derive_relationship(*variable_follower*, *variable_leaders*, *required_scenario='*'*, *required_model='*'*, *interpkind='linear'*)

Derive the relationship between two variables from the database.

Parameters

- **variable_follower** (*str*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CH4"`).
- **variable_leaders** (*list[str]*) – The variable(s) we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **required_scenario** (*str or list[str]*) – The string(s) which all relevant scenarios are required to match. This may have *s to represent wild cards. It defaults to* “” to accept all scenarios.
- **required_model** (*str or list[str]*) – The string(s) which all relevant models are required to match. This may have *s to represent wild cards. It defaults to* “” to accept all models.
- **interpkind** (*str*) – The style of interpolation. By default, linear, but can also be any value accepted as the “kind” option in scipy.interpolate.interp1d, or “PchipInterpolator”, in which case scipy.interpolate.PchipInterpolator is used. Care should be taken if using non-default interp1d options, as they are either uneven or have “ringing”

Returns

Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type

`func`

Raises

ValueError – There is no data of the appropriate type in the database. There may be a typo in the SSP option.

4.5 Latest Time Ratio API

Module for the database cruncher which uses the ‘latest time ratio’ technique.

`class silicone.database_crunchers.latest_time_ratio.`**`LatestTimeRatio`**`(db)`

Bases: `_DatabaseCruncher`

Database cruncher which uses the ‘latest time ratio’ technique.

This cruncher derives the relationship between two variables by simply assuming that the follower timeseries is equal to the lead timeseries multiplied by a scaling factor. The scaling factor is derived by calculating the ratio of the follower variable to the lead variable in the latest year in which the follower variable is available in the database. Additionally, since the derived relationship only depends on a single point in the database, no regressions or other calculations are performed.

Once the relationship is derived, the 'filler' function will infill following:

$$E_f(t) = R * E_l(t)$$

where $E_f(t)$ is emissions of the follower variable and $E_l(t)$ is emissions of the lead variable, both in the infillee database.

R is the scaling factor, calculated as

$$R = \frac{E_f(t_{\text{last}})}{e_l(t_{\text{last}})}$$

where t_{last} is the average of all values of the follower gas at the latest time it appears in the database, and the lower case e represents the infiller database.

derive_relationship(*variable_follower*, *variable_leaders*)

Derive the relationship between two variables from the database.

Parameters

- **variable_follower** (*str*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|C5F12"`).
- **variable_leaders** (*list[str]*) – The variable we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`). Note that the 'latest time ratio' methodology gives the same result, independent of the value of `variable_leaders` in the database.

Returns

Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type

`func`

Raises

- `ValueError` – `variable_leaders` contains more than one variable.
- `ValueError` – There is no data for `variable_leaders` or `variable_follower` in the database.

4.6 Linear interpolation cruncher API

Module for the database cruncher which makes a linear interpolator between known values

class `silicone.database_crunchers.linear_interpolation.`**LinearInterpolation**(*db*)

Bases: `Interpolation`

Database cruncher which uses linear interpolation. This cruncher is deprecated; use Interpolation instead.

This cruncher derives the relationship between two variables by linearly interpolating between values in the cruncher database. It does not do any smoothing and is best-suited for smaller databases.

In the case where there is more than one value of the follower variable for a given value of the leader variable, the average will be used. For example, if one scenario has CH4 emissions of 10 MtCH4/yr whilst another has

CH4 emissions of 20 MtCH4/yr in 2020 whilst both scenarios have CO2 emissions of exactly 15 GtC/yr in 2020, the interpolation will use the average value from the two scenarios i.e. 15 Mt CH4/yr.

Beyond the bounds of input data, the interpolation is held constant. For example, if the maximum CO2 emissions in 2020 in the database is 25 GtC/yr, and CH4 emissions for this level of CO2 emissions are 15 MtCH4/yr, then even if we infill using a CO2 emissions value of 100 GtC/yr in 2020, the returned CH4 emissions will be 15 MtCH4/yr.

`derive_relationship``(`*variable_follower*, *variable_leaders*`)`

Derive the relationship between two variables from the database.

Parameters

- **`variable_follower`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CH4"`).
- **`variable_leaders`** (*`list[str]`*) – The variable(s) we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **`interpkind`** (*`str`*) – The style of interpolation. By default, linear (hence the name), but can also be any value accepted as the "kind" option in scipy.interpolate.interp1d.

Returns

Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type

`func`

Raises

`ValueError` – There is no data of the appropriate type in the database.

4.7 Quantile rolling windows cruncher API

Module for the database cruncher which uses the 'rolling windows' technique.

`class` `silicone.database_crunchers.quantile_rolling_windows.`**`QuantileRollingWindows`**`(`*db*`)`

Bases: `_DatabaseCruncher`

Database cruncher which uses the 'rolling windows' technique.

This cruncher derives the relationship between two variables by performing quantile calculations between the follower timeseries and the lead timeseries. These calculations are performed at each timestep in the timeseries, independent of the other timesteps.

For each timestep, the lead timeseries axis is divided into multiple evenly spaced windows (to date this is only tested on 1:1 relationships but may work with more than one lead timeseries). In each window, every data point in the database is included. However, the data points receive a weight given by

$$w(x, x_{\text{window}}) = \frac{1}{1 + (d_n)^2}$$

where w is the weight and d_n is the normalised distance between the centre of the window and the data point's position on the lead timeseries axis.

d_n is calculated as

$$d_n = \frac{x - x_{\text{window}}}{f \times (\frac{b}{2})}$$

where x is the position of the data point on the lead timeseries axis, x_{window} is the position of the centre of the window on the lead timeseries axis, b is the distance between window centres and f is a decay factor which controls how much less points away from x_{window} are weighted. If $f = 1$ then a point which is half the width between window centres away receives a weighting of $1/2$. Lowering the value of f cause points further from the window centre to receive less weight.

With these weightings, the desired quantile of the data is then calculated. This calculation is done by sorting the data by the database's follow timeseries values (then by lead timeseries values in the case of identical follow values). From here, the weight of each point is calculated following the formula given above. We calculate the cumulative sum of weights, and then the cumulative sum up to half weights, defined by

$$c_{hw} = c_w - 0.5 \times w$$

where c_w is the cumulative weights and w is the raw weights. This ensures that quantiles less than half the weight of the smallest follow value return the smallest follow value and more than one minus half the weight of the largest follow value return the largest value. Without such a shift, the largest value is only returned if the quantile is 1, leading to a bias towards smaller values.

With these calculations, we have determined the relationship between the follow timeseries values and the quantile i.e. cumulative sum of (normalised) weights. We can then determine arbitrary quantiles by linearly interpolating.

If the option `use_ratio` is set to `True`, instead of returning the absolute value of the follow at this quantile, we return the quantile of the ratio between the lead and follow data in the database, multiplied by the actual lead value of the database being infilled.

By varying the quantile, this cruncher can provide ranges of the relationship between different variables. For example, it can provide the 90th percentile (i.e. high end) of the relationship between e.g. `Emissions|CH4` and `Emissions|CO2` or the 50th percentile (i.e. median) or any other arbitrary percentile/quantile choice. Note that the impact of this will strongly depend on nwindows and decay_length_factor. Using the `TimeDepQuantileRollingWindows` class makes it is possible to specify a dictionary of dates to quantiles, in which case we return that quantile for that year or date.

`derive_relationship`(*variable_follower*, *variable_leaders*, *quantile=0.5*, *nwindows=11*, *decay_length_factor=1*, *use_ratio=False*)

Derive the relationship between two variables from the database.

Parameters

- **`variable_follower`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CH4"`).
- **`variable_leaders`** (*`list[str]`*) – The variable(s) we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **`quantile`** (*`float`*) – The quantile to return in each window.
- **`nwindows`** (*`int`*) – The number of window centers to use when calculating the relationship between the follower and lead gases.
- **`decay_length_factor`** (*`float`*) – Parameter which controls how strongly points away from the window's centre should be weighted compared to points at the centre. Larger values give points further away increasingly less weight, smaller values give points further away increasingly more weight.
- **`use_ratio`** (*`bool`*) – If false, we use the quantile value of the weighted mean absolute value. If true, we find the quantile weighted mean ratio between lead and follow, then multiply the ratio by the input value.

Returns
Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type
`func`

Raises

- `ValueError` – There is no data for `variable_leaders` or `variable_follower` in the database.
- `ValueError` – `quantile` is not between 0 and 1.
- `ValueError` – `nwindows` is not equivalent to an integer or is not greater than 1.
- `ValueError` – `decay_length_factor` is 0.

4.8 Time dependent quantile rolling windows cruncher API

Module for the database cruncher which uses the 'rolling windows' technique with different quantiles in different years.

`class silicone.database_crunchers.time_dep_quantile_rolling_windows.`**`TimeDepQuantileRollingWindows`**(*db*)

Bases: `_DatabaseCruncher`

Database cruncher which uses QuantileRollingWindows with different quantiles in every year/datetime.

`derive_relationship`(*variable_follower*, *variable_leaders*, *time_quantile_dict*, ***kwargs*)

Derive the relationship between two variables from the database.

For details of most parameters, see QuantileRollingWindows. The one different parameter is time_quantile_dict, described below:

Parameters

- **`variable_follower`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CH4"`).
- **`variable_leaders`** (*`list[str]`*) – The variable(s) we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **`time_quantile_dict`** (*`dict{datetime or int:  float}`*) – Every year or datetime in the infillee database must be specified as a key. The value is the quantile to use in that year. Note that the impact of the quantile value is strongly dependent on the arguments passed to `QuantileRollingWindows`.
- **`**kwargs`** – Passed to `QuantileRollingWindows`.

Returns
Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type
`func`

Raises
`ValueError` – Not all times in `time_quantile_dict` have data in the database.

4.9 Time dependent ratio cruncher API

Module for the database cruncher which uses the 'time-dependent ratio' technique.

class `silicone.database_crunchers.time_dep_ratio.`**`TimeDepRatio`**(*db*)

Bases: `_DatabaseCruncher`

Database cruncher which uses the 'time-dependent ratio' technique.

This cruncher derives the relationship between two variables by simply assuming that the follower timeseries is equal to the lead timeseries multiplied by a time-dependent scaling factor. The scaling factor is the ratio of the follower variable to the lead variable. If the database contains many such pairs, the scaling factor is the ratio between the means of the values. By default, the calculation will include only values where the lead variable takes the same sign (+ or -) in the infilling database as in the case infilled. This prevents getting negative values of emissions that cannot be negative. To allow cases where we have no data of the correct sign, set *same_sign = False* in *derive_relationship*.

Once the relationship is derived, the 'filler' function will infill following:

$$E_f(t) = R(t) * E_l(t)$$

where $E_f(t)$ is emissions of the follower variable and $E_l(t)$ is emissions of the lead variable.

$R(t)$ is the scaling factor, calculated as the ratio of the means of the the follower and the leader in the infiller database, denoted with lower case e. By default, we include only cases where *sign(e_l(t))* is the same in both databases). The cruncher will raise a warning if the lead data is ever negative, which can create complications for the use of this cruncher.

$$R(t) = \frac{mean(e_f(t))}{mean(e_l(t))})$$

`derive_relationship`(*variable_follower*, *variable_leaders*, *same_sign=True*, *only_consistent_cases=True*)

Derive the relationship between two variables from the database.

Parameters

- **`variable_follower`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|C5F12"`).
- **`variable_leaders`** (*`list[str]`*) – The variable we want to use in order to infer timeseries of `variable_follower` (e.g. `["Emissions|CO2"]`).
- **`same_sign`** (*`bool`*) – Do we want to only use data where the leader has the same sign in the infiller and infillee data? If so, we have a potential error from not having data of the correct sign, but have more confidence in the sign of the follower data.
- **`only_consistent_cases`** (*`bool`*) – Do we want to only use model/scenario combinations where both lead and follow have data at all times? This will reduce the risk of inconsistencies or unevenness in the results, but will slightly decrease performance speed if you know the data is consistent. Senario/model pairs where data is only returned at certain times will be removed, as will any scenarios not returning both lead and follow data.

Returns
Function which takes a `pyam.IamDataFrame` containing `variable_leaders` timeseries and returns timeseries for `variable_follower` based on the derived relationship between the two. Please see the source code for the exact definition (and docstring) of the returned function.

Return type
`func`

Raises

- `ValueError` – `variable_leaders` contains more than one variable.
- `ValueError` – There is no data for `variable_leaders` or `variable_follower` in the database.

CHAPTER

FIVE

MULTIPLE INFILLERS API

Multiple infillers

Multiple infillers provide easy-to-use infiller options for the most common use-cases.

5.1 Decompose collection with time-dependent ratio API

Uses the 'time-dependent ratio' database cruncher designed for constructing an aggregate variable and breaking this mix into its constituents.

`class silicone.multiple_infillers.decompose_collection_with_time_dep_ratio.DecomposeCollectionTimeDepRa`

Bases: `object`

Constructs an aggregate variable and uses the 'time-dependent ratio' technique to calculate what this predicts for our database.

`infill_components`(*aggregate*, *components*, *to_infill_df*, *metric_name='AR5GWP100'*, *only_consistent_cases=True*)

Derive the relationship between the composite variables and their sum, then use this to deconstruct the sum.

Parameters

- **aggregate** (*str*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CO2"`). Unlike in most crunchers, we do not expect the database to already contain this data.
- **components** (*list[str]*) –

  The variables whose sum should be equal to the timeseries of the aggregate
  (e.g. `["Emissions|CO2|AFOLU", "Emissions|CO2|Energy"]`).
- **to_infill_df** (`pyam.IamDataFrame`) – The dataframe that already contains the `aggregate` variable, but needs the `components` to be infilled.
- **metric_name** (*str*) – The name of the conversion metric to use. This will usually be AR<4/5/6>GWP100.
- **only_consistent_cases** (*bool*) – Do we want to only use model/scenario combinations where all aggregate and components have data at all times? This will reduce the risk of inconsistencies or unevenness in the results, but may reduce the amount of data.

Returns
The infilled data resulting from the calculation.

Return type
`pyam.IamDataFrame`

Raises
`ValueError` – There is no data for `variable_leaders` or `variable_follower` in the database.

5.2 Infill all required emissions for openscm API

`silicone.multiple_infillers.infill_all_required_emissions_for_openscm.`**`infill_all_required_variables`**(*to_fi
data
vari-
able
re-
quir
crun
'sil-
i-
cone
out-
put_
in-
filled
to_fi
chec
**kw*

This is function designed to infill all required data given a minimal amount of input.

Parameters

- **`to_fill`** (`pyam.IamDataFrame`) – The dataframe which is to be infilled
- **`database`** (`pyam.IamDataFrame`) – The dataframe containing all information to be used in the infilling process.
- **`variable_leaders`** (*`list[str]`*) – The name of the variable(s) found in to_fill which should be used to determine the values of the other variables. For most infillers (including the default) this list must contain only one entry. E.g. [“Emissions|CO2”]
- **`required_variables_list`** (*`list[str]`*) – The list of variables to infill. Each will be done separately. The default behaviour (None option) will result in this being filled with the complete list of required emissions.
- **`cruncher`** – The class of cruncher to use to compute the infilled values. Defaults to QuantileRollingWindows, which uses the median value of a rolling window. See the cruncher documentation for more details.
- **`output_timesteps`** (*`list[int or datetime]`*) – List of times at which to return infilled values. Will interpolate values in between known data, but will not extend beyond the range of data provided.
- **`infilled_data_prefix`** (*`str`*) – A string that should be prefixed on all the variable names of the results returned. Used to distinguish returned values from those input.
- **`to_fill_old_prefix`** (*`str`*) – Any string already found at the beginning of the variables names of the input *to_fill* dataframe. This will be removed before comparing the variable names with *database*.

- **check_data_returned** (*bool*) – If true, we perform checks that all desired data has been returned. Potential reasons for failing this include requesting results at times outside our input time range, as well as code bugs.
- **kwargs** (*******) – An optional dictionary of keyword : arguments to be used with the cruncher.

Returns
The infilled dataframe (including input data) at requested times. All variables now begin with infilled_data_prefix instead of to_fill_old_prefix.

Return type
`pyam.IamDataFrame`

5.3 Infill composite values API

`silicone.multiple_infillers.infill_composite_values.`**`infill_composite_values`**(*df*, *composite_dic=None*)

Constructs a series of aggregate variables, calculated as the sums of variables that have been reported. If given factors terms too, the terms will be multiplied by the factors before summing.

Parameters

- **df** (`pyam.IamDataFrame`) – Input data from which to construct consistent values. This is assumed to be fully infilled. This will not be checked.
- **composite_dic** (*dict{str: list[str]} or dict{str: dict{str: float}}*) – Key: The variable names of the composite. Value: The variable names of the constituents, which may include wildcards ('*'). Optionally, these values may be dictionaries of the names to factors, which we multiply the numbers by before summing them. Defaults to a list of PFC, HFC, F-Gases, CO2 and Kyoto gases.

Returns
`pyam.IamDataFrame` containing the composite values.

Return type
`pyam.IamDataFrame`

CHAPTER

SIX

TIME PROJECTORS API

Time projectors

The classes in this module are used to infer values for a scenario at later times given the trends before that time.

6.1 Extend latest time quantile

Module for the database cruncher which uses the 'latest time quantile' technique.

class silicone.time_projectors.extend_latest_time_quantile.**ExtendLatestTimeQuantile**(*db*)

Bases: `object`

Time projector which extends the timeseries of a variable by assuming that it remains that a fixed quantile in the infiller database, the quantile it is in at the last available time. This is the natural counterpart to the equal quantile walk extending a single variable over time rather than over different emissions.

It assumes that the target timeseries is shorter than the infiller timeseries.

derive_relationship(*variable*, *smoothing=None*, *weighting=None*)

Derives the quantiles of the variable in the infiller database. Note that this takes only one variable as an argument, whereas most crunchers take two.

Parameters

- **variable** (*str*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CO2"`).
- **smoothing** (*float or string*) – By default, no smoothing is done on the distribution. If a value is provided, it is fed into `scipy.stats.gaussian_kde()` - see full documentation there. In short, if a float is input, we fit a Gaussian kernel density estimator with that width to the points. If a string is used, it must be either "scott" or "silverman", after those two methods of determining the best kernel bandwidth.
- **weighting** (*None or dict{(str, str): float}*) – The dictionary, mapping the (model and scenario) tuple onto the weight (relative to a weight of 1 for the default). This does not have to include all scenarios in df, but cannot include scenarios not in df.

Returns

Function which takes a `pyam.IamDataFrame` containing `variable` timeseries and returns these timeseries extended until the latest time in the infiller database.

Return type

`func`

Raises

`ValueError` – There is no data for `variable` in the database.

6.2 Extend RMS closest

Module for the database cruncher that uses the rms closest extension method

class `silicone.time_projectors.extend_rms_closest.`**`ExtendRMSClosest`**(*db*)

Bases: `object`

Time projector which extends the timeseries of a variable with future timesteps infilled using the values from the 'closest' pathway in the infilling database.

We define the closest pathway as the pathway with the smallest time-averaged (over the reported time steps) root mean squared difference

`derive_relationship`(*variable*)

Derives the values for the model/scenario combination in the database with the least RMS error.

Parameters

`variable` (*`str`*) – The variable for which we want to calculate the timeseries (e.g. *Emissions|CO2*).

Returns

Filled in data (without original source data)

Return type

obj: *pyam.IamDataFrame*

6.3 Linear extender

Module for the database cruncher which extends using a linear trend

class `silicone.time_projectors.linear_extender.`**`LinearExtender`**(*db=None*)

Bases: `object`

Time projector which extends the timeseries of a variable using a linear trend. You can either specify a gradient for the line (possibly zero) or a point in the future.

`derive_relationship`(*variable*, *gradient=None*, *year_value=None*, *times=None*)

Derives the function to return a linear trend following from the last datapoint

Parameters

- **`variable`** (*`str`*) – The variable for which we want to calculate timeseries (e.g. `"Emissions|CO2"`).
- **`gradient`** (*`float or None`*) – The gradient of the variable after its last available datapoint, in the emissions units per year. If not provided, year_value must be provided instead.
- **`year_value`** (*`None or tuple(int or datetime, float)`*) – The value of the variable at a given future time, e.g. (2050, 0) to extend the data to net zero in 2050. If not provided, gradient must be provided instead.
- **`times`** (*`None or list[int or datetime]`*) – The times to return entries at. Only required if no database was used during initalisation.

Returns

Function which takes a `pyam.IamDataFrame` containing `variable` timeseries and returns these timeseries extended until the latest time in the infiller database.

Return type
`func`

Raises
`ValueError` – There is no data for `variable` in the database.

CHAPTER SEVEN

STATS API

Silicone's custom statistical operations.

silicone.stats.**calc_all_emissions_correlations**(*emms_df*, *years*, *output_dir*)

Save csv files of the correlation coefficients and the rank correlation coefficients between emissions at specified times.

This function includes all undivided emissions (i.e. results recorded as *Emissions|X*) and CO2 emissions split once (i.e. *Emissions|CO2|X*). It does not include Kyoto gases. It will also save the average absolute value of the coefficients.

Parameters

- **emms_df** (`pyam.IamDataFrame`) – The database to search for correlations between named values
- **output_dir** (*str*) – The folder location to save the files.
- **years** (*list[int]*) – The years upon which to calculate correlations.
- **created** (*Files*) –
- **-------------** –
- **"variable_counts.csv"** (*the number of scenario/model pairs where the emissions*) –
- **occurs.** (*data*) –
- **"gases_correlation_{year}.csv"** (*The Pearson's correlation between gases emissions*) – in a given year.
- **"gases_rank_correlation_{year}.csv"** (*The Spearman's rank correlation between*) –
- **year** (*gases in a given*) –
- **"time_av_absolute_correlation_{}_to_{}.csv"** (*The magnitude of the Pearson's*) –
- **emissions** (*correlation between*) –
- **requested.** (*averaged over the years*) –
- **"time_av_absolute_rank_correlation_{}_to_{}.csv"** (*The magnitude of the Spearman's*) – rank correlation between emissions, averaged over the years requested.
- **"time_variance_rank_correlation_{}_to_{}.csv"** (*The variance over time in the rank*) – correlation values above.

silicone.stats.**calc_quantiles_of_data**(*distribution*, *points_to_quant*, *smoothing=None*, *weighting=None*, *to_quantile=True*)

Calculates the quantiles of points_to_quant in the distribution of values described by distribution. Optionally treats points_to_quant as quantiles and returns the values that would lead to them instead.

Parameters

- **distribution** (*pd.Series*) – The distribution of values.
- **points_to_quant** (*pd.Series*) – The points which we want find: if `to_quantile` is `True` (default) these are the values which we will compare to the distribution, if `False`, these are the quantiles which we want to find.
- **smoothing** (*float or string*) – By default, no smoothing is done on the distribution. If a value is provided, it is fed into `scipy.stats.gaussian_kde()` - see full documentation there. In short, if a float is input, we fit a Gaussian kernel density estimator with that width to the points. If a string is used, it must be either "scott" or "silverman", after those two methods of determining the best kernel bandwidth.
- **weighting** (*None or Series*) – If a series, must have the same indices as distribution, giving the relative weights of each point.
- **to_quantile** (*Bool*) – If `True`, we return the quantiles of the data in points_to_quant. If `False`, we instead treat points_to_quant as the quantiles themselves (they must all be 0-1) and return the values in distribution that occur at these quantiles.

Returns

An array with one row and a column for each entry in points_to_quant, containing the quantiles of these points in order. Or, if to_quantile is `False`, containing the values corresponding to the quantiles points_to_quant.

Return type

np.ndarray

silicone.stats.**rolling_window_find_quantiles**(*xs*, *ys*, *quantiles*, *nwindows=11*, *decay_length_factor=1*)

Perform quantile analysis in the y-direction for x-weighted data.

Divides the x-axis into nwindows of equal length and weights data by how close they are to the center of these windows. Then returns the quantiles of this weighted data. Quantiles are defined so that the values returned are always equal to a y- value in the data - there is no interpolation. Extremal points are given their full weighting, meaning this will not agree with the np.quantiles under uniform weighting (which effectively gives 0 weight to min and max values).

The weighting of a point at x for a window centered at x_0 is:

$$w = \frac{1}{1 + \left(\frac{x - x_0}{l_{window}} \times f_{dl}\right)^2}$$

for l_{window} the window width (range of values divided by nwindows -1) and f_{dl} the decay_length_factor.

Parameters

- **xs** (np.ndarray, `pd.Series`) – The x co-ordinates to use in the regression.
- **ys** (np.ndarray, `pd.Series`) – The y co-ordinates to use in the regression.
- **quantiles** (*list-like*) – The quantiles to calculate in each window
- **nwindows** (*int*) – How many points to evaluate between x_max and x_min. Must be > 1.
- **decay_length_factor** (*float*) – gives the distance over which the weighting of the values falls to 1/4, relative to half the distance between window centres. Defaults to 1.

Returns
Quantile values at the window centres.

Return type
`pd.DataFrame`

Raises
`AssertionError` – `xs` and `ys` don’t have the same shape

CHAPTER

EIGHT

UTILS API

silicone.utils.**convert_units_to_MtCO2_equiv**(*df*, *metric_name='AR5GWP100'*)

Converts the units of gases reported in kt into Mt CO2 equivalent per year

Uses GWP100 values from either (by default) AR5 or AR4 IPCC reports.

Parameters

- **df** (pyam.IamDataFrame) – The input dataframe whose units need to be converted.
- **metric_name** (*str*) – The name of the conversion metric to use. This will usually be AR<4/5/6>GWP100.

Returns
The input data with units converted.

Return type
pyam.IamDataFrame

silicone.utils.**download_or_load_sr15**(*filename*, *valid_model_ids='*'*)

Load SR1.5 data, if it isn't there, download it

Parameters

- **filename** (*str*) – Filename in which to look for/save the data
- **valid_model_ids** (*str*) – Models to return from data

Returns
The loaded data

Return type
obj: *pyam.IamDataFrame*

silicone.utils.**find_matching_scenarios**(*options_df*, *to_compare_df*, *variable_follower*, *variable_leaders*, *classify_scenarios*, *classify_models=['*']*, *return_all_info=False*, *use_change_not_abs=False*)

Groups scenarios and models into different classifications and uses those to work out which group contains a trendline most similar to the data. These combinations may group several models/scenarios together by means of wild cards. Most similar means having the smallest total squared distance between the to_compare_df value of variable_follower and the variable_follower values interpolated in options_df at the variable_leaders points in to_compare_df, i.e. assuming errors only exist in variable_follower. In the event of a tie between different scenario/model classifications, it returns the scenario/model combination that occurs earlier in the input lists, looping through scenarios first.

Parameters

- **options_df** (pyam.IamDataFrame) – The dataframe containing the data for each scenario and model

- **to_compare_df** (pyam.IamDataFrame) – The dataframe we wish to find the scenario group closest to. May contain one or more scenarios, we minimise the least squared errors for all the data colleectively.
- **variable_follower** (*str*) – The variable we want to interpolate and compare to the value in to_compare_df
- **variable_leaders** (*list[str]*) – The variable(s) we want to use to construct the interpolation (e.g. `["Emissions|CO2"]`). In the event that there are multiple, we interpolate with each one separately and minimise the sum of the squared errors.
- **classify_scenarios** (*list[str]*) – The names of scenarios or groups of scenarios that are possible matches. This may have "*"s to represent wild cards, hence multiple scenarios will have all their data combined to make the interpolator.
- **classify_models** (*list[str]*) – The names of models or groups of models that are possible matches. This may have "*"s to represent wild cards, hence multiple models will have all their data combined to make the interpolator.
- **return_all_info** (*bool*) – If True, instead of simply returning the strings specifying the closest scenario/model match, we return all scenario/model combinations in order of preference, along with the rms distance, quantifying the closeness.
- **use_change_not_abs** (*bool*) – If True, the code looks for the trend with the closest *derivatives* rather than the closest absolute value, i.e. closest trend allowing for an offset. This requires data from more than one time.

Returns

- *if return_all_info == False*
- *(string, string)* – Strings specifying the model (first) and scenario (second) classifications that best match the data.
- *if return_all_info == True*
- *dict* – Maps the model and scenario classification strings to the measure of closeness.

Raises

ValueError – Not all required timepoints are present in the database we crunched, we have *{dates we have}* but you passed in *{dates we need}*."

silicone.utils.**get_sr15_scenarios**(*output_file*, *valid_model_ids*)

Collects world-level data from the IIASA database for the named models and saves them to a given location.

Parameters

- **output_file** (*str*) – File name and location for data to be saved
- **valid_model_ids** (*list[str]*) – Names of models that are to be fetched.

silicone.utils.**return_cases_which_consistently_split**(*df*, *aggregate*, *components*, *how_close=None*, *metric_name='AR5GWP100'*)

Returns model-scenario tuples which correctly split up the to_split into the various components. Components may contain wildcard "*"s to match several variables.

Parameters

- **df** (pyam.IamDataFrame) – The input dataframe.
- **aggregate** (*str*) – Name of the variable that should split into the others
- **components** (*list[str]*) – List of the variable names whose sum should equal the to_split value (if expressed in common units).

- **how_close** (*dict*) – This is a dictionary of numpy.isclose options specifying how exact the match must be for the case to be included as passing. By default we specify a relative tolerance of 1% ('rtol': 1e-2). The syntax for this can be found in the numpy documentation.
- **metric_name** (*str*) – The name of the conversion metric to use. This will usually be AR<4/5/6>GWP100.

Returns
List of consistent (Model name, scenario name, region name) tuples.

Return type
list[(str, str, str)]

CHAPTER NINE

CHANGELOG

9.1 [v1.3.0] - 14 Oct 2022

9.1.1 Added

- (#146) Added the ability to do non-linear interpolation by introducing `Interpolation`.

9.1.2 Changed

- (#149) Checked consistency of time extender database before extending at constant quantiles.
- (#142) Sped up RMS closest
- (#146) Deprecated the linear interpolator `LinearInterpolation` in favour of a generic interpolator `Interpolation`.
- (#150) Added more info when returning an error message in multiple infillers.

9.1.3 Fixed

- (#149) Ensured RMS closest works with the latest version of pyam. Bugfix for a warning in infill_composite_values
- (#144) RMS closest no longer causes `pd.core.common.SettingWithCopyWarning` to be raised
- (#147) Filter prevents including data from the wrong regions in `DecomposeCollectionTimeDepRatio`. Notebook fixed to run with updates in python 3.8.

9.2 [v1.2.0] - 28 Sept 2021

9.2.1 Added

- (#139) Support for pyam-iamc>1.0
- (#135) Added html documentation of the time projectors

9.2.2 Changed

- (#138) Remove Python3.6 support
- (#138) Improved speed of `silicone.multiple_infillers.infill_all_required_emissions_for_openscm()` by removing multiple loops (note that API did not change)

9.3 [v1.1.0] - 12 July 2021

9.3.1 Added

- (#134) Added Gaurav to author list.
- (#132) Added an additional time projector (Linear Extender) that simply extends the latest data to reach a specified point or by a constant gradient.
- (#129) Added an additional time projector (Extend RMS closest) that extends a pathway to cover later times by selecting future data from the closest pathway.
- (#126) Added the first time projector (Extend latest time quantile) that extends a pathway to cover later times, assuming it remains at the same quantile.

9.3.2 Changed

- (#133) More fixes to allow compatibility with pyam updates.
- (#131) Updated to allow compatibility with latest versions of pyam, openscm-units, coverage, pytest and black

9.3.3 Fixed

- (#130) Reformatted files to make the linter happy (no functional changes).

9.4 [v1.0.3]

9.4.1 Changed

- (#124) Neatened up the changelog

9.5 [v1.0.2] - 4 Jan 2021

9.5.1 Changed

- (#121) Updated to openscm-units>0.2

9.5.2 Fixed

- (#123) Made the installation runner avoid prerelease.

9.6 [v1.0.1] - 27 Oct 2020

9.6.1 Added

- (#115) Enabled multiple lead gases to be used with RMS closest cruncher.

9.6.2 Changed

- (#119) Updated to work with pyam v0.8

9.7 [v1.0.0] - 9 Sept 2020

9.7.1 Initial release

- (#116) Pinned black
- (#113) Added a warning for using ratio-based crunchers with negative values. Fixed some unit conversion todos (not user-facing).
- (#112) Enabled more general unit conversion, bug fix and improvement for infill_composite_values.
- (#111) Minor improvements to error messages and documentation.
- (#110) Gave an option to time_dep_ratio and decompose_collection to ignore model/scenario combinations missing values at some required times.
- (#108) Added a multiple infiller to split up an aggregate with a remainder. Disabled test for downloading database.
- (#103) Update github address to GranthamImperial.
- (#101) Update release docs
- (#93) Add regular test of install from PyPI
- (#102) Minor bugfix for nan handling in Equal Quantile Walk.
- (#100) Added funding info to readme and removed unnecessary files.
- (#97) Added sections to documentation file so that newer crunchers and multiple infillers are included.
- (#95) Added sections to notebooks covering all the recent changes.
- (#94) Added `EqualQuantileWalk`, a cruncher which finds the quantile of the lead variable in the infiller database and returns the same quantile of the follow variable.
- (#87) Added `TimeDepQuantileRollingWindows`, a cruncher which allows the user to crunch different quantiles in different years.
- (#86) Slightly changed the definition of quantile rolling windows to make it symmetric (not rounding down).
- (#83) Added tests for appending results of crunching to the input.
- (#82) Updated to a later version of pyam and solved todos associated with this. Also added a `kwargs` argument to `infill_all_required`.

- (#80) Changed the names of crunchers for brevity. Also changed `lead_gas` to `latest_time_ratio` and included it in ratio notebook.
- (#78) Changed how quantile rolling windows works by adding an extra interpolate step for smoothness
- (#77) Added calculation of variance of rank correlation to stats
- (#76) Removed command-line interface
- (#75) Updated README
- (#72) Altered infill_composite_value to allow multiplication by a factor before summing. Removed unnecessary notebooks.
- (#69) Fixed bug so that `DatabaseCruncherRMSClosest` no longer selects scenarios which don't have follower data
- (#68) More investigatory tools and scripts for calculating and outputting emissions correlations.
- (#67) Introduce investigatory tools for plotting relations between emissions.
- (#66) Remove `Input` folder in favour of using `openscm-units`
- (#65) Add `format-notebooks` target to the `Makefile`
- (#64) Add basic linters to CI
- (#61) Switch to using GitHub actions for CI
- (#60) Update installation docs to reference pip and conda
- (#62) Minor changes to remove warning messages and remove some todos.
- (#52) Made the Lead Gas infiller use the average latest data rather than being restricted to a single value. Updated infill_composite_values to work with the latest data.
- (#51) Split the notebooks into chapters with minor changes to the text. Moved a script function into utilities to download data.
- (#49) Rewrote the documentation and notebooks to update, split up information and clarify.
- (#48) Introduced multiple_infiller function to calculate the composite values from the constituents.
- (#47) Made an option for quantile_rolling_windows to infill using the ratio of lead to follow data.
- (#46) Made the time-dependent ratio infiller only use data where the leader has the same sign.
- (#45) Made infill_all_required_emissions_for_openscm, the second multiple-infiller function.
- (#44) Made decompose_collection_with_time_dep_ratio, the first multiple-infiller function.
- (#43) Implemented new util functions for downloading data, unit conversion and data checking.
- (#41) Added a cruncher to interpolate values between data from specific scenarios. Only test notebooks with lax option.
- (#32) Raise *ValueError* when asking to infill a case with no data
- (#27) Developed the constant ratio cruncher
- (#21) Developed the time-dependent ratio cruncher
- (#20) Clean up the quantiles cruncher and test rigorously
- (#19) Add releasing docs plus command-line entry point tests
- (#14) Add root-mean square closest pathway cruncher
- (#13) Get initial work (see #11) into package structure, still requires tests (see #16)

- (#12) Add BSD-3-Clause license
- (#9) Add lead gas cruncher
- (#6) Update development docs
- (#5) Put notebooks under CI
- (#4) Add basic documentation structure
- (#1) Added pull request and issues templates

CHAPTER TEN

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX