FaIR

Release 2.1.0

FaIR Development Team

Dec 19, 2022

CONTENTS

FaIR is a reduced-complexity climate model useful for scenario assessment and idealised climate runs.

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 From the Python Package Index (PyPI)

Probably the easiest way to get up and running:

```
pip install fair
```

1.1.2 From GitHub

Download and install

The latest release can be obtained from https://github.com/OMS-NetZero/FAIR/releases as zip or tarball files, or the most current unreleased version can be cloned from https://github.com/OMS-NetZero/FAIR.

Developing

1. Fork the repository, then clone it to your local disk. *cd* to the *FAIR* directory in your working copy.
2. Create a new branch for your changes:

   ```
   git checkout -b <branch_name>
   ```

3. Optional, but we highly recommend developing FaIR in a virtual environment to keep your base installation of *python* nice and clean.
4. Install *fair* in development mode:

   ```
   pip install -e .[dev]
   ```

5. Make your changes.
6. Write a test that tests your new feature (in the `tests` directory of the repository).
7. Format your code, and run tests locally:

   ```
   make format
   make checks
   make tests
   make test_notebooks
   ```

If you find errors at this point, they will need fixing before GitHub will allow merging to the *master* branch. Running the test suite ensures that your code change does not break or change existing functionality.

8. Commit and push your changes:

```
git add <file>
git commit -m "informative commit message"
git push origin <branch_name>
```

9. Create a pull request on the parent repository to merge in your changes. You will see there's a checklist of processes to go through…
10. One of which is adding a line to *CHANGELOG.rst* with a summary of the changes made.
11. The checks and tests will run using GitHub actions. If all pass, you should be able to submit the pull request for review.
12. If the codeowners are happy, the branch will be merged in.

TODO: Check out the (currently non-existent) contributing guide, but it's basically this.

1.1.3 Conda install

Coming soon.

1.2 Introduction

This introduction addresses some of the key concepts of FaIR and how they are now implemented inside the model. For worked examples of FaIR and the energy balance model, take a look at the *Examples*.

FaIR produces global mean temperature projections from various forcers. Input datasets can be provided in terms of emissions, concentrations (for greenhouse gases), or effective radiative forcing. Different species can be input in different ways, if they are internally consistent and valid (e.g. if you want to provide CO_2 and short-lived climate forcer emissions, with concentrations of non-CO_2 greenhouse gases and prescribed forcing for volcanic and solar forcing, like the *esm-hist* runs of CMIP6 Earth System Models). You can provide any number of species from single-forcing runs to full climate assessments. In "full-climate" mode, the process diagram in FaIR is similar to Fig. 1.1.

FaIR is now a primarily object-oriented interface. The main interaction with FaIR is through the `FAIR` class instance:

```
from fair import FAIR
f = FAIR()
```

`FAIR()` contains the attributes and methods required to run the model. The energy balance climate model can also be run in standalone mode with prescribed forcing. In the rest of this introduction, `f` is taken to be a `FAIR` instance.

Typically there are two stages to a FaIR run. The first is to set up the dimensionality of the problem (define time horizon, included species and how to implement them), and the second is to fill in the data and do the run.

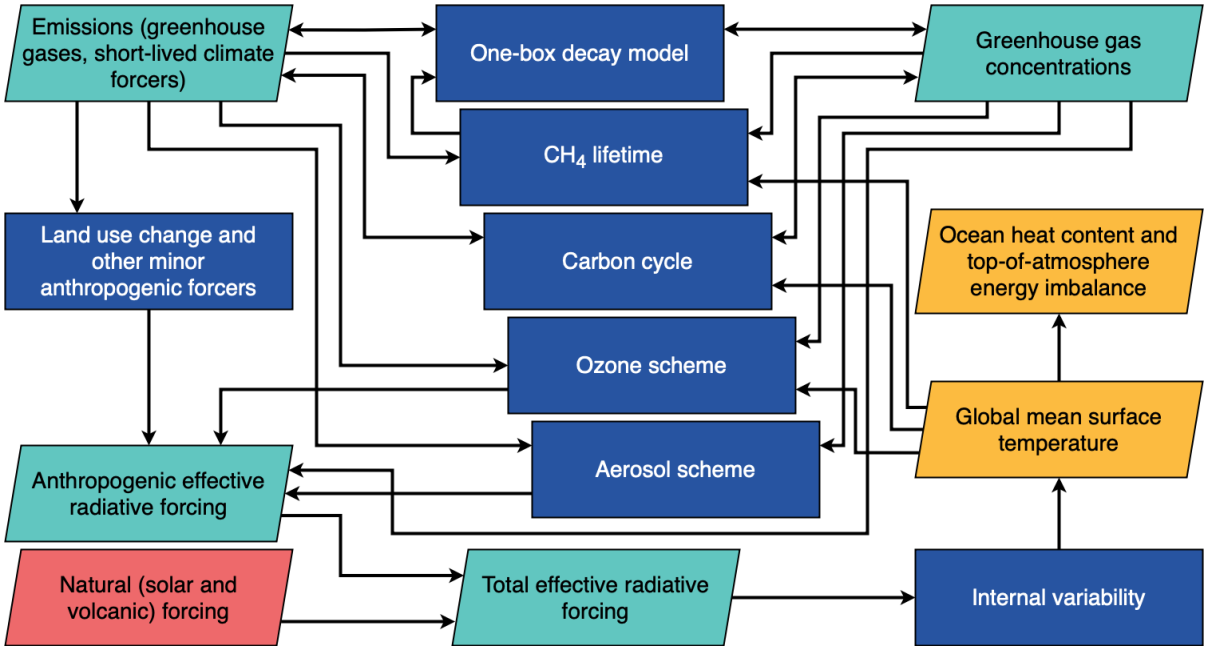

Fig. 1.1: Simplified outline of available processes in FaIR. Some processes have been omitted for clarity.

1.2.1 Dimensionality

Internally, variables in are book-kept with `numpy` arrays that have up to five dimensions:

[time, scenario, config, specie, box/layer]

Not all variables include all dimensions, but they are always in this order.

Parallelisation allows more than one `scenario` and one set of `configs` to be run at the same time. `time` is the only axis which is looped over, allowing high efficiency multiple-scenario or multiple-configuration runs to be done on the same processor (subject to available RAM and disk space).

For inputs and outputs, we use `xarray`, which wraps the `numpy` arrays with axis labels and makes model input and output a little more tractable.

Defining the dimensionality of the problem is the first step of setting up FaIR.

Time

FaIR uses two time variables to keep track of state variables: `timepoints` and `timebounds`. Only emissions are defined on `timepoints`, where everything else is on `timebounds`. As the name suggests, `timebounds` are on the boundary of model time steps, and `timepoints` are nominally at the centre of the model time step. Fig. 1.2 illustrates this.

Time is specified in years, but you are not forced to use integer years. This can be useful for coupling directly with integrated assessment model derived emissions (often 5- or 10-year timesteps) or assessing short-term responses to volcanic forcing, where sub-annual responses can be important. Fig. 1.3 shows the timestepping for a $1/4$-year timestep.

A FaIR ecosystem therefore contains $n/\delta t$ `timepoints` and $n/\delta t + 1$ `timebounds` where δt is the timestep and n is the number of years.

Time is defined as so:

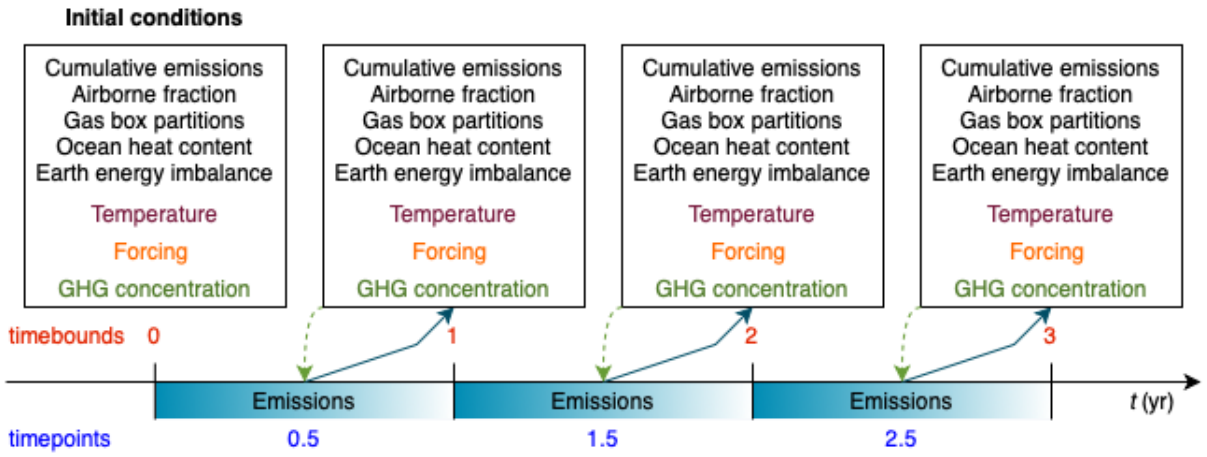

Fig. 1.2: Timebounds (red) and timepoints (blue) in FaIR, showing associated state variables defined on each time variable. FaIR can be run with prescibed emissions (blue arrows) and/or with prescribed concentrations for greenhouse gases (green arrows).

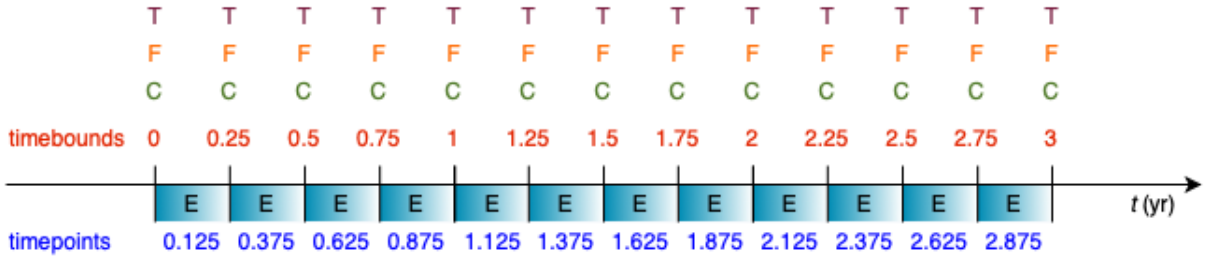

Fig. 1.3: Timebounds (red) and timepoints (blue) in FaIR using a quarter-year time step.

```
f.define_time(start, end, timestep)
```

You can label time how you wish. Common runs for climate projections will start with a pre-industrial reference year (maybe 1750 or 1850), and run to 2100 or beyond:

```
f.define_time(1750, 2100, 1)
```

Or you might want to run an idealised pulse experiment, in which case it's less informative to assign real years and it is better to start at zero:

```
f.define_time(0, 1000, 1)
```

The simple non-integer example from Fig. 1.3 would thus be:

```
f.define_time(0, 3, 0.25)
```

We adopt the `pandas` practice of including the end point.

For intepreting real dates, the `timebound` 1750 refers to 00:00 on 1 January 1750, and the `timepoint` 1750.5 is a mid-year average of the period 00:00 on 1 January 1750 to 00:00 on 1 January 1751. FaIR makes no adjustments for leap years or month length, and assumes each year is 365.24219 days long. The temporal difference between `timebound` and `timepoint` can be important when interpreting and reporting results: does temperature in 2100 mean at the `timebound` 2100, or do we want a mid-year average (corresponding to `timepoint` 2100.5, in which case we might want to interpolate or take an average of the 2100 and 2101 `timebounds`.)

Scenarios

A `scenario` is a set of emissions/concentration/forcing inputs and climate responses. Multiple `scenarios` can be run in parallel. They are defined as a list of names, for example:

```
f.define_scenarios(['ssp126', 'ssp245', 'ssp370'])
```

Note at this stage we are only defining names: no data is being input into FaIR.

Configs

A `config` defines a set of climate and species parameters to run FaIR with. For example, we might want to run with emulations of a few CMIP6 models, which have different climate sensitivities, aerosol forcing sensitivities to precurors, carbon cycle feedback strengths, and so on:

```
f.define_configs(['UKESM', 'NorESM', 'GFDL', 'MIROC'])
```

Again, we are only defining names at this stage.

Combined with the three `scenarios` above, we have a matrix of 12 runs: each of three emissions scenarios will be run with each of four climate configs.

Species

A `specie` is anything that forces climate or other species in FaIR, which is a broad definition. This includes greenhouse gases, aerosol precurors, ozone precursors as expected, but also forcing categories (aerosol-radiation interactions, ozone forcing, land-use forcing) that may be calculated from other species.

Each `specie` has an associated dict of `properties` which defines how it is implemented in the particular run of FaIR and how it behaves. A `properties` dict for a CO_2 run might look like:

```
properties = {
    'CO2': {
        'type': 'co2',
        'input_mode': 'emissions',
        'greenhouse_gas': True,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    }
}
```

The five dict keys of `type`, `input_mode`, `greenhouse_gas`, `aerosol_chemistry_from_emissions` and `aerosol_chemistry_from_concentration` are all required. Both `type` and `input_mode` are from pre-defined lists. The API reference for fair.FAIR.define_species explains more.

Species are then declared in FaIR with:

```
f.define_species(['CO2'], properties)
```

Basic checks are performed to ensure that input specifications make sense (e.g. you cannot run a type `solar` in `emissions` mode, and many species types must be unique). An error will be raised if an invalid combination of options is provided.

For all but single-forcing experiments, defining `species` this way could be quite onerous. To grab an emissions-driven run using all `species` known to FaIR with their default values, use:

```
species, properties = read_properties()
f.define_species(species, properties)
```

Box and layer

FaIR uses a multiple-box atmospheric decay model with lifetime scaling for greenhouse gases (see e.g. [Millar2017]). This is represented by the `box` dimension. By default, there are 4 boxes, but this can be modified in the initialisation:

```
f = FAIR(n_gasboxes=3)
```

or by accessing the attribute directly after initialisation:

```
f.n_gasboxes=3
```

`layer` refers to the ocean layer of the energy balance model. By default, FaIR uses 3 layers, though this can be modified in the initialisation of the class:

```
f = FAIR(n_layers=2)
```

or by accessing the attribute directly:

```
f.n_layers=3
```

1.2.2 State variables

State variables are attributes of the `FAIR` class. All state variables are outputs, and many are valid inputs (particularly for the first `timebound` in which many must be provided with an initial condition).

After problem setup (see ref:*Dimensionality*), these `xarrays` will be created inside FaIR with:

```
f.allocate()
```

Dimensions with invalid combinations are retained in the output (e.g. `f.emissions` will be `np.nan` for solar forcing) to maintain alignment between datasets.

Emissions

Emissions are the only variable defined on `timepoints`, and are quantified as an emissions rate per year. FaIR will automatically adjust the emissions flows if a non- annual timestep is provided. Emissions are input or output as the `emissions` attribute of `FAIR`:

```
f.emissions :: [timepoint, scenario, config, specie]
```

Concentrations

For greenhouse gases, concentrations can be input, or calculated:

```
f.concentration :: [timebound, scenario, config, specie]
```

Note the FaIR variable name is `concentration`, in the singular.

When running in emissions mode, the initial concentration of a greenhouse gas should be specified (at `timebound` 0). We provide a convenience function, `initialise()`, for specifying initial conditions:

```
from fair import initialise
initialise(f.concentration, 278.3, specie='CO2')
```

You could also directly modify the `f.concentration` `xarray` by label:

```
f.concentration.loc[dict(timebound=1750, specie='CO2')] = 278.3
```

or position, if you know that CO_2 is index 0 of the *specie* axis:

```
f.concentration[0, :, :, 0] = 278.3
```

Effective radiative forcing

At the per-species level, effective radiative forcing can be input or calculated:

```
f.forcing :: [timebound, scenario, config, specie]
```

We use the shorter name `forcing`, which should be taken to represent effective radiative forcing.

For some species like solar and volcanic, specifying forcing is the only valid input mode.

Again, initial forcing must be provided. In many cases, this will be zero for every species, in which case we don't have to specify it for each `specie`:

```
initialise(f.forcing, 0)
```

The total effective radiative forcing is an output only:

```
f.forcing_sum :: [timebound, scenario, config]
```

and is simply `forcing` summed over the `specie` axis.

Units are W m^{-2}.

Temperature

Temperatures in FaIR are expressed as anomalies (in units of Kelvin) relative to some reference state, usually pre-industrial. Temperature is calculated from an n-layer ocean energy balance model:

```
f.temperature :: [timebound, scenario, config, layer]
```

The temperature near the surface and hence of most importance is layer 0. Again, initial conditions of temperature in all layers should be provided, and for a "cold-start" model can be done with:

```
initialise(f.temperature, 0)
```

One of the most important sources of climate projection uncertainty is the temperature response to forcing, which is governed by the `climate_config` parameters (see later). These are varied across the `config` dimension in FaIR. Simply, these define how much, and how quickly, temperature change occurs in response to a given forcing and encapsulates *ECS* and *TCR* which are emergent parameters.

Airborne emissions

Airborne emissions are the total stock of a `specie` present in the atmosphere and usually expressed as an anomaly relative to pre-industrial:

```
f.airborne_emissions :: [timebound, scenario, config, specie]
```

Again, `airborne_emissions` should be initialised. For "warm-start" runs, airborne emissions may be non-zero, and this value has influence on the carbon and methane cycles in FaIR.

Airborne fraction

Airborne fraction is the fraction of airborne emissions remaining in the atmosphere:

```
f.airborne_fraction :: [timebound, scenario, config, specie]
```

It is simply `f.airborne_emissions` divided by `f.cumulative_emissions`. It does not need to be initialised.

Cumulative emissions

The cumulative emissions are the summed emissions since pre-industrial:

```
f.cumulative_emissions :: [timebound, scenario, config, specie]
```

`cumulative_emissions` needs to be initialised. It is used in the carbon cycle and land use forcing components of FaIR.

Ocean heat content

The ocean heat content change is the time integral of the top of atmosphere energy imbalance (after some unit conversion):

```
f.ocean_heat_content_change :: [timebound, scenario, config]
```

Units are J. Divide by 10^{21} to get the more common zettajoules (ZJ) unit. It does not need to be initialised.

Stochastic forcing

If using stochastic internal variability, this is the stochastic component of the total effective radiative forcing (see [Cummins2020]). Its dimensionality is:

```
f.stochastic_forcing :: [timebound, scenario, config]
```

Units are W m^{-2}. It does not need to be initialised.

Top of atmosphere energy imbalance

Follows from the energy balance model:

```
f.toa_imbalance :: [timebound, scenario, config]
```

Units are W m^{-2}. It does not need to be initialised.

Gas box partition fractions

The partition fraction is the only state variable where the `timebound` is not carried, which would necessitate a 5-dimensional array and use up useful memory. So we have:

```
f.gasbox_fractions :: [scenario, config, specie, gasbox]
```

1.2.3 Inputting variable data

Once the state variables have been allocated, we need to fill our input variables with data. Any valid method for inputting data into `xarray` can be used. Additionally, we have a second convenience function `fill()` that can do this for us:

```
from fair import fill
fill(f.emissions, 38, specie="CO2", scenario="ssp370")
```

Browse the *Examples* for instances where the variables are filled in.

1.2.4 Adjusting configs

For each defined `config` (in our earlier example, we used the names from four CMIP6 models), we can (and should!) vary the model response. This gives us our climate uncertainty. There are two types: `climate_configs` and `species_configs`. As with the state variables, they are attributes of the FAIR instance, and implemented as `xarray` Datasets.

Climate configs

`climate_configs` define the behaviour of the energy balance model and contain the following variables:

- ocean_heat_transfer
- ocean_heat_capacity
- deep_ocean_efficacy
- forcing_4co2
- stochastic_run
- sigma_eta
- sigma_xi
- gamma_autocorrelation
- seed

They can be filled in for example using:

```
fill(f.climate_configs["ocean_heat_capacity"], [2.92, 11.28, 73.25], config='UKESM')
```

The API reference for the energy_balance_model explains more.

Species configs

`species_configs` define the behaviour of individual species: most variables have a `species` dimension, some also have a `gasbox` dimension. It contains the following variables:

- tropospheric_adjustment
- forcing_efficacy
- forcing_temperature_feedback
- forcing_scale
- partition_fraction

- unperturbed_lifetime
- molecular_weight
- baseline_concentration
- iirf_0
- iirf_airborne
- iirf_uptake
- iirf_temperature
- baseline_emissions
- g0
- g1
- concentration_per_emission
- forcing_reference_concentration
- greenhouse_gas_radiative_efficiency
- contrails_radiative_efficiency
- erfari_radiative_efficiency
- h2o_stratospheric_factor
- lapsi_radiative_efficiency
- land_use_cumulative_emissions_to_forcing
- ozone_radiative_efficiency
- aci_scale
- aci_shape
- cl_atoms
- br_atoms
- fractional_release
- ch4_lifetime_chemical_sensitivity
- lifetime_temperature_sensitivity

There are a *lot* of things that can be modified here. If you want to use default values for your defined `species`, you can use:

```
f.fill_species_configs()
```

To override a default (or to fill it in if you didn't use `fill_species_configs()`), modify the `xarray` directly or use `fill()`:

```
fill(f.species_configs["aci_shape"], 0.0370, config='UKESM', specie='Sulfur')
```

1.2.5 Run

As simple as:

```
f.run()
```

The outputs are stored in the `xarray` attributes. For example, if you ran emissions-driven, you should (if everything worked) find concentrations in `f.concentration` calculated for species that were declared as greenhouse gases. And vice versa: FaIR will automatically back-calculate emissions where greenhouse gas concentrations are provided, if you set up your run this way.

If you find unexpected NaNs in your outputs, it's likely that something wasn't initialised in the above steps. Take a look at the *Examples* for workflows that should reproduce. If you can't figure it out and think it should work, raise an issue.

1.2.6 Glossary

ECS
: The equilibrium climate sensitivity (ECS) is the long-term equilibrium global mean surface temperature response to a doubling of pre-industrial atmospheric CO_2 concentrations with all other forcers fixed at pre-industrial. It is a measure of long-term sensitivity of climate.

TCR
: The transient climate response (TCR) is the global mean surface temperature change at the point where double pre-industrial atmospheric CO_2 concentration is reached in an experiment where CO_2 concentration is increased at a compound rate of 1% per year and all other forcers fixed at pre-industrial. This is around 69.7 years (many models use 70 years). It is a measure of both long-term sensitivity of climate and medium-term rate of climate response.

1.3 Examples

1.3.1 Basic example

FaIR v2.1 is object-oriented and designed to be more flexible than its predecessors. This does mean that setting up a problem is different to before - gone are the days of 60 keyword arguments to `fair_scm` and we now use classes and functions with fewer arguments that in the long run should be easier to use. Of course, there is a learning curve, and will take some getting used to. This tutorial aims to walk through a simple problem using FaIR 2.1.

The structure of FaIR 2.1 centres around the `FAIR` class, which contains all information about the scenario(s), the forcer(s) we want to investigate, and any configurations specific to each species and the response of the climate.

A run is initialised as follows:

```
f = FAIR()
```

To this we need to add some information about the time horizon of our model, forcers we want to run with, their configuration (and the configuration of the climate), and optionally some model control options:

```
f.define_time(2000, 2050, 1)
f.define_scenarios(['abrupt', 'ramp'])
f.define_configs(['high', 'central', 'low'])
f.define_species(species, properties)
f.ghg_method='Myhre1998'
```

We generate some variables: emissions, concentrations, forcing, temperature etc.:

```
f.allocate()
```

which creates xarray DataArrays that we can fill in:

```
fill(f.emissions, 40, scenario='abrupt', specie='CO2 FFI')
...
```

Finally, the model is run with

```
f.run()
```

Results are stored within the FAIR instance as xarray DataArrays or Dataset, and can be obtained such as

```
print(fair.temperature)
```

Multiple scenarios and configs can be supplied in a FAIR instance, and due to internal parallelisation is the fastest way to run the model (100 ensemble members per second for 1750-2100 on my Mac for an emissions driven run). The total number of scenarios that will be run is the product of scenarios and configs. For example we might want to run three emissions scenarios – let's say SSP1-2.6, SSP2-4.5 and SSP3-7.0 – using climate calibrations (configs) from the UKESM, GFDL, MIROC and NorESM climate models. This would give us a total of 12 ensemble members in total which are run in parallel.

The most difficult part to learning FaIR 2.1 is correctly defining the scenarios and configs. As in previous versions of FaIR, there is a lot of flexibility, and simplifying the calling interface (fair_scm in v1.x) has come at the cost of switching this around to the FAIR class, and things have to be defined in the right order usually.

Recommended order for setting up a problem

In this tutorial the recommended order in which to define a problem is set out step by step, and is as follows:

1. Create the FAIR instance, inititalised with run control options.
2. Define the time horizon of the problem with FAIR.define_time()
3. Define the scenarios to be run (e.g. SSPs, IAM emissions scenarios, or anything you want) with FAIR.define_scenarios().
4. Define the configurations to be run with FAIR.define_configs(). A configuration (config) is a set of parameters that describe climate response and species response parameters. For example you might have a config with high climate sensitivity and strong aerosol forcing, and one with low climate sensitivity and weak aerosol forcing.
5. Define which species will be included in the problem, and their properties including the run mode (e.g. emissions-driven, concentration driven) with FAIR.define_species().
6. Optionally, modify run control options.
7. Create input and output DataArrays with FAIR.allocate().
8. Fill in the DataArrays (e.g. emissions), climate configs, and species configs, by either working directly with the xarray API, or using FaIR-packaged convenience functions like fill and initialise.
9. Run: FAIR.run().
10. Analyse results by accessing the DataArrays that are attributes of FAIR.

1. Create FaIR instance

We'll call our instance `f`: it's nice and short and the `fair` name is reserved for the module.

```
from fair import FAIR
```

```
f = FAIR()
```

2. Define time horizon

There are two different time indicators in FaIR: the `timebound` and the `timepoint`. `timebound`s, as the name suggests, are at the edges of each time step; they can be thought of as instantaneous snapshots. `timepoint`s are what happens between time bounds and are rates or integral quantities.

The main thing to remember is that only `emissions` are defined on `timepoint`s and everything else is defined on `timebound`s, and when we specify the time horizon in our model, we are defining the `timebound`s of the problem.

Secondly, the number of `timebound`s is one more than the number of `timepoint`s, as the start and end points are included in the `timebound`s.

```
# create time horizon with bounds of 2000 and 2050, at 1-year intervals
f.define_time(2000, 2050, 1)
print(f.timebounds)
print(f.timepoints)
```

3. Define scenarios

The scenarios are a list of strings that label the scenario dimension of the model, helping you keep track of inputs and outputs.

In this example problem we will create two scenarios: an "abrupt" scenario (where emissions or concentrations change instantly) and a "ramp" scenario where they change gradually.

```
# Define two scenarios
f.define_scenarios(["abrupt", "ramp"])
f.scenarios
```

4. Define configs

Similarly to the scenarios, the configs are a labelling tool. Each config has associated climate- and species-related settings, which we will come to later.

We'll use three config sets, crudely corresponding to high, medium and low climate sensitivity.

```
# Define three scenarios
f.define_configs(["high", "central", "low"])
f.configs
```

5. Define species

This defines the forcers – anthropogenic or natural – that are present in your scenario. A `species` could be something directly emitted like CO2 from fossil fuels, or it could be a category where forcing has to be calculate from precursor emissions like aerosol-cloud interactions.

Each `specie` is assigned a name that is used to distinguish it from other species. You can call the species what you like within the model as long as you are consistent. We also pass a dictionary of `properties` that defines how each specie behaves in the model.

In this example we'll start off running a scenario with CO2 from fossil fuels and industry, CO2 from AFOLU, CH4, N2O, and Sulfur (note you don't need the full 40 species used in v1.1-1.6, and some additional default ones are included). From these inputs we also want to determine forcing from aerosol-radiation and aerosol-cloud interactions, as well as CO2, CH4 and N2O.

To highlight some of the functionality we'll run CO2 and Sulfur emissions-driven, and CH4 and N2O concentration-driven. (This is akin to an `esm-ssp585` kind of run from CMIP6, though with fewer species). We'll use totally fake data here - this is not intended to represent a real-world scenario but just to highlight how FaIR works. Full simulations may have 50 or more species included and the `properties` dictionary can get quite large, so it can be beneficial to edit it in a CSV and load it in.

In total, we have 8 species in this model. We want to run

1. CO2 fossil and industry
2. CO2 AFOLU
3. Sulfur

with specified emissions.

We want to run

4. CH4
5. N2O

with specified concentrations. We also want to calculate forcing from CO2, so we need to declare the CO2 as a greenhouse gas in addition to its emitted components:

6. CO2

and we want to calculate forcing from aerosol radiation and aerosol cloud interactions

7. ERFari
8. ERFaci

```
species = ['CO2 fossil emissions', 'CO2 AFOLU emissions', 'Sulfur', 'CH4', 'N2O', 'CO2',
↪'ERFari', 'ERFaci']
```

In the `properties` dictionary, the keys must match the `species` that you have declared. I should do another tutorial on changing some of the properties; but

- `type` defines the species type such as CO2, an aerosol precursor, or volcanic forcing; there's around 20 predefined types in FaIR. Some can only be defined once in a run, some can have multiple instances (e.g. `f-gas`). See `fair.structure.species` for a list.
- `input_mode`: how the model should be driven with this `specie`. Valid values are `emissions`, `concentration`, `forcing` or `calculated` and not all options are valid for all `type`s (e.g. running solar forcing with concentrations). `calculated` means that the emissions/concentration/forcing of this specie depends on others, for example aerosol radiative forcing needs precursors to be emitted.

- `greenhouse_gas`: True if the `specie` is a greenhouse gas, which means that an associated `concentration` can be calculated (along with some other species-specific behaviours). Note that CO2 emissions from fossil fuels or from AFOLU are not treated as greenhouse gases.
- `aerosol_chemistry_from_emissions`: Some routines such as aerosols, methane lifetime, or ozone forcing, relate to emissions of short-lived climate forcers. If this `specie` is one of these, this should be set to True.
- `aerosol_chemistry_from_concentration`: As above, but if the production of ozone, aerosol etc. depends on the concentration of a greenhouse gas.

```
properties = {
    'CO2 fossil emissions': {
        'type': 'co2 ffi',
        'input_mode': 'emissions',
        'greenhouse_gas': False,  # it doesn't behave as a GHG itself in the model, but␣
↪as a precursor
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    },
    'CO2 AFOLU emissions': {
        'type': 'co2 afolu',
        'input_mode': 'emissions',
        'greenhouse_gas': False,  # it doesn't behave as a GHG itself in the model, but␣
↪as a precursor
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    },
    'CO2': {
        'type': 'co2',
        'input_mode': 'calculated',
        'greenhouse_gas': True,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    },
    'CH4': {
        'type': 'ch4',
        'input_mode': 'concentration',
        'greenhouse_gas': True,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': True, # we treat methane as a reactive␣
↪gas
    },
    'N2O': {
        'type': 'n2o',
        'input_mode': 'concentration',
        'greenhouse_gas': True,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': True, # we treat nitrous oxide as a␣
↪reactive gas
    },
    'Sulfur': {
        'type': 'sulfur',
        'input_mode': 'emissions',
        'greenhouse_gas': False,
```

(continues on next page)

(continued from previous page)

```
        'aerosol_chemistry_from_emissions': True,
        'aerosol_chemistry_from_concentration': False,
    },
    'ERFari': {
        'type': 'ari',
        'input_mode': 'calculated',
        'greenhouse_gas': False,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    },
    'ERFaci': {
        'type': 'aci',
        'input_mode': 'calculated',
        'greenhouse_gas': False,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    }
}
```

```
f.define_species(species, properties)
```

6. Modify run options

When we initialise the FAIR class, a number of options are given as defaults.

Let's say we want to change the greenhouse gas forcing treatment from Meinshausen et al. 2020 to Myhre et al. 1998. While this could have been done when initialising the class, we can also do it by setting the appropriate attribute.

```
help(f)
```

```
f.ghg_method
```

```
f.aci_method='myhre1998'
```

```
f.aci_method
```

7. Create input and output data

Steps 2–5 above dimensioned our problem; now, we want to actually create some data to put into it.

First we allocate the data arrays with

```
f.allocate()
```

This has created our arrays with the correct dimensions as attributes of the `FAIR` class:

```
f.emissions
```

```
f.temperature
```

8. Fill in the data

The data created is nothing more special than `xarray` DataArrays, and using `xarray` methods we can allocate values to the emissions:

```
f.emissions.loc[(dict(specie="CO2 fossil emissions", scenario="abrupt"))] = 38
```

```
f.emissions[:,0,0,0]
```

I think this method is a tiny bit clunky with `loc` and `dict` so two helper functions have been created; `fill` and `initialise`. It's personal preference if you use them or not, the only thing that matters is that the data is there.

```
from fair.interface import fill, initialise
```

8a. fill emissions, concentrations ...

Remember that some species in our problem are emissions driven, some are concentration driven, and you might have species which are forcing driven (though not in this problem).

You will need to populate the datasets to ensure that all of the required species are there, in their specified driving mode.

```
import numpy as np
```

```
fill(f.emissions, 38, scenario='abrupt', specie='CO2 fossil emissions')
fill(f.emissions, 3, scenario='abrupt', specie='CO2 AFOLU emissions')
fill(f.emissions, 100, scenario='abrupt', specie='Sulfur')
fill(f.concentration, 1800, scenario='abrupt', specie='CH4')
fill(f.concentration, 325, scenario='abrupt', specie='N2O')

for config in f.configs:
    fill(f.emissions, np.linspace(0, 38, 50), scenario='ramp', config=config, specie=
↪'CO2 fossil emissions')
    fill(f.emissions, np.linspace(0, 3, 50), scenario='ramp', config=config, specie='CO2␣
↪AFOLU emissions')
    fill(f.emissions, np.linspace(2.2, 100, 50), scenario='ramp', config=config, specie=
↪'Sulfur')
    fill(f.concentration, np.linspace(729, 1800, 51), scenario='ramp', config=config,␣
↪specie='CH4')
    fill(f.concentration, np.linspace(270, 325, 51), scenario='ramp', config=config,␣
↪specie='N2O')
```

We also need approriate initial conditions. If you are seeing a lot of unexpected NaNs in your results, it could be that the first timestep was never defined.

Using non-zero values for forcing, temperature, airborne emissions etc. such as from the end of a previous run may allow for restart runs in the future.

```
# Define first timestep
initialise(f.concentration, 278.3, specie='CO2')
initialise(f.forcing, 0)
initialise(f.temperature, 0)
```

(continues on next page)

(continued from previous page)

```
initialise(f.cumulative_emissions, 0)
initialise(f.airborne_emissions, 0)
```

8b. Fill in climate_configs

This defines how the model responds to a forcing: the default behaviour is the three-layer energy balance model as described in Cummins et al. (2020). The number of layers can be changed in run_control.

climate_configs is an xarray Dataset.

```
f.climate_configs
```

```
fill(f.climate_configs["ocean_heat_transfer"], [0.6, 1.3, 1.0], config='high')
fill(f.climate_configs["ocean_heat_capacity"], [5, 15, 80], config='high')
fill(f.climate_configs["deep_ocean_efficacy"], 1.29, config='high')

fill(f.climate_configs["ocean_heat_transfer"], [1.1, 1.6, 0.9], config='central')
fill(f.climate_configs["ocean_heat_capacity"], [8, 14, 100], config='central')
fill(f.climate_configs["deep_ocean_efficacy"], 1.1, config='central')

fill(f.climate_configs["ocean_heat_transfer"], [1.7, 2.0, 1.1], config='low')
fill(f.climate_configs["ocean_heat_capacity"], [6, 11, 75], config='low')
fill(f.climate_configs["deep_ocean_efficacy"], 0.8, config='low')
```

8c. Fill in species_configs

This is again an xarray Dataset, with lots of options. Most of these will be made loadable defaults, and indeed you can load up defaults with

FAIR.fill_species_configs()

For this example we'll show the manual editing of the species configs, which you will probably want to do anyway in a full run (e.g. to change carbon cycle sensitivities).

```
f.species_configs
```

Greenhouse gas state-dependence

iirf_0 is the baseline time-integrated airborne fraction (usually over 100 years). It can be calculated from the variables above, but sometimes we might want to change these values.

```
fill(f.species_configs["partition_fraction"], [0.2173, 0.2240, 0.2824, 0.2763], specie=
↪"CO2")

non_co2_ghgs = ["CH4", "N2O"]
for gas in non_co2_ghgs:
    fill(f.species_configs["partition_fraction"], [1, 0, 0, 0], specie=gas)

fill(f.species_configs["unperturbed_lifetime"], [1e9, 394.4, 36.54, 4.304], specie="CO2")
```

(continues on next page)

(continued from previous page)

```
fill(f.species_configs["unperturbed_lifetime"], 8.25, specie="CH4")
fill(f.species_configs["unperturbed_lifetime"], 109, specie="N2O")

fill(f.species_configs["baseline_concentration"], 278.3, specie="CO2")
fill(f.species_configs["baseline_concentration"], 729, specie="CH4")
fill(f.species_configs["baseline_concentration"], 270.3, specie="N2O")

fill(f.species_configs["forcing_reference_concentration"], 278.3, specie="CO2")
fill(f.species_configs["forcing_reference_concentration"], 729, specie="CH4")
fill(f.species_configs["forcing_reference_concentration"], 270.3, specie="N2O")

fill(f.species_configs["molecular_weight"], 44.009, specie="CO2")
fill(f.species_configs["molecular_weight"], 16.043, specie="CH4")
fill(f.species_configs["molecular_weight"], 44.013, specie="N2O")

fill(f.species_configs["greenhouse_gas_radiative_efficiency"], 1.3344985680386619e-05,␣
↪specie='CO2')
fill(f.species_configs["greenhouse_gas_radiative_efficiency"], 0.00038864402860869495,␣
↪specie='CH4')
fill(f.species_configs["greenhouse_gas_radiative_efficiency"], 0.00319550741640458,␣
↪specie='N2O')
```

```
# some greenhouse gas parameters can be automatically calculated from lifetime,␣
↪molecular weight and partition fraction:
f.calculate_iirf0()
f.calculate_g()
f.calculate_concentration_per_emission()
```

```
# but we still want to override sometimes, and because it's just an xarray, we can:
fill(f.species_configs["iirf_0"], 29, specie='CO2')
```

```
# Now we define sensitivities of airborne fraction for each GHG; I'll do this quickly
fill(f.species_configs["iirf_airborne"], [0.000819*2, 0.000819, 0], specie='CO2')
fill(f.species_configs["iirf_uptake"], [0.00846*2, 0.00846, 0], specie='CO2')
fill(f.species_configs["iirf_temperature"], [8, 4, 0], specie='CO2')

fill(f.species_configs['iirf_airborne'], 0.00032, specie='CH4')
fill(f.species_configs['iirf_airborne'], -0.0065, specie='N2O')

fill(f.species_configs['iirf_uptake'], 0, specie='N2O')
fill(f.species_configs['iirf_uptake'], 0, specie='CH4')

fill(f.species_configs['iirf_temperature'], -0.3, specie='CH4')
fill(f.species_configs['iirf_temperature'], 0, specie='N2O')
```

Aerosol emissions or concentrations to forcing

Note, both here and with the GHG parameters above, we don't have to change parameters away from NaN if they are not relevant, e.g. Sulfur is not a GHG so we don't care about `iirf_0`, and CO2 is not an aerosol precursor so we don't care about `erfari_radiative_efficiency`.

```
fill(f.species_configs["erfari_radiative_efficiency"], -0.0036167830509091486, specie=
↪'Sulfur') # W m-2 MtSO2-1 yr
fill(f.species_configs["erfari_radiative_efficiency"], -0.002653/1023.2219696044921,␣
↪specie='CH4') # W m-2 ppb-1
fill(f.species_configs["erfari_radiative_efficiency"], -0.00209/53.96694437662762,␣
↪specie='N2O') # W m-2 ppb-1

fill(f.species_configs["aci_scale"], -2.09841432)
fill(f.species_configs["aci_shape"], 1/260.34644166, specie='Sulfur')
```

9. run FaIR

```
f.run()
```

10. plot results

```
import matplotlib.pyplot as pl
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ramp', layer=0)], label=f.configs)
pl.title('Ramp scenario: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
pl.legend()
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ramp', specie='CO2')], label=f.
↪configs)
pl.title('Ramp scenario: CO2')
pl.xlabel('year')
pl.ylabel('CO2 (ppm)')
pl.legend()
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ramp', specie='ERFaci')], label=f.
↪configs)
pl.title('Ramp scenario: forcing')
pl.xlabel('year')
pl.ylabel('ERF from aerosol-cloud interactions (W m$^{-2}$)')
pl.legend()
```

```
pl.plot(f.timebounds, f.forcing_sum.loc[dict(scenario='ramp')], label=f.configs)
pl.title('Ramp scenario: forcing')
pl.xlabel('year')
pl.ylabel('Total ERF (W m$^{-2}$)')
pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='abrupt', layer=0)], label=f.
↪configs)
pl.title('Abrupt scenario: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
pl.legend()
```

```
pl.plot(f.timebounds, f.forcing_sum.loc[dict(scenario='abrupt')], label=f.configs)
pl.title('Abrupt scenario: forcing')
pl.xlabel('year')
pl.ylabel('Total ERF (W m$^{-2}$)')
pl.legend()
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='abrupt', specie='CO2')],␣
↪label=f.configs)
pl.title('Abrupt scenario: CO2')
pl.xlabel('year')
pl.ylabel('CO2 (ppm)')
pl.legend()
```

```
f.species_configs['g0'].loc[dict(specie='CO2')]
```

```
f.forcing[-1, :, 1, :]
```

1.3.2 SSP emissions runs using CMIP6 calibrations

This notebook gives an example of running the SSP scenarios in FaIR using pre-calculated climate response calibrations from 66 CMIP6 models for a total of 8 × 66 = 528 scenarios in parallel. We will run in stochastic mode to demonstrate internal variability.

This demonstrates the speed in which FaIR can run scenarios and configurations in parallel. This might be a good template notebook to use for SSP emissions-driven runs.

Refer to the `basic_emissions_run_example` for more detail on setting up a run step by step; this notebook will be a little lighter on the detail, but hopefully still enough to be useful.

0. import required modules

```
import numpy as np
import matplotlib.pyplot as pl
import pandas as pd

from fair import FAIR
from fair.io import read_properties
from fair.interface import fill, initialise
from fair.earth_params import seconds_per_year
```

1. initialise FaIR

We want to enable the methane lifetime routine that is a function of SLCFs and reactive gases, i.e. using the coefficients and feedbacks from Thornhill et al. 2021 and Skeie et al. 2020. We set this option in the initialiser this time.

```
f = FAIR(ch4_method='thornhill2021')
```

2. Define time horizon

create world running from 1750 to 2100, at 1-year intervals

```
f.define_time(1750, 2100, 1)
```

3. Define scenarios

We'll take the SSPs; important that the names are consistent with those in the RCMIP database

```
# Define SSP scenarios
scenarios = ['ssp119', 'ssp126', 'ssp245', 'ssp370', 'ssp434', 'ssp460', 'ssp534-over',
↪'ssp585']
f.define_scenarios(scenarios)
```

4. Define configs

Our list of configs are going to be each CMIP6 climate model's 4xCO2 response, which has been pre-calculated in the calibration notebooks.

We could also modify the response for different aerosol, ozone, methane lifetime tunings etc., but not every model has this data available.

```
df = pd.read_csv("../tests/test_data/4xCO2_cummins_ebm3.csv")
models = df['model'].unique()
configs = []

for imodel, model in enumerate(models):
    for run in df.loc[df['model']==model, 'run']:
        configs.append(f"{model}_{run}")
f.define_configs(configs)
```

5. Define species and properties

FaIR contains a few helper functions that populate the model with sensible defaults. One is the `read_properties` function that obtains default species (the kitchen sink) and their properties for an emissions-driven run

```
species, properties = read_properties()
#species = list(properties.keys())
```

```
species[:5]
```

```
properties['CO2 FFI']
```

```
f.define_species(species, properties)
```

~6. Modify run options~

Not required, because we set our run options in step 1.

7. Create input and output data

```
f.allocate()
```

8. Fill in the data

8a. get default species configs

Again we read in a default list of species configs that will apply to each config. If you want to change specific configs then you can still use this function to set defaults and tweak what you need. We will do this with the methane lifetime, which has a different value calibrated for the Thornhill 2021 lifetime option.

I'm also going to subtract the RCMIP 1750 emissions from CH4 and N2O. This is not in the default configs.

```
f.fill_species_configs()
fill(f.species_configs['unperturbed_lifetime'], 10.8537568, specie='CH4')
fill(f.species_configs['baseline_emissions'], 19.01978312, specie='CH4')
fill(f.species_configs['baseline_emissions'], 0.08602230754, specie='N2O')
```

8b. fill emissions

grab emissions (+solar and volcanic forcing) from RCMIP datasets using the `fill_from_rcmip` helper function. This function automatically selects the emissions, concentration or forcing you want depending on the `properties` for each of the SSP scenarios defined.

I'm then going to make one change: replace the volcanic dataset with the AR6 volcanic dataset, as I want to compare the impact of monthly volcanic forcing in the monthly comparison.

We also need to initialise the first timestep of the run in terms of its per-species forcing, temperature, cumulative and airborne emissions. We set these all to zero. The concentration in the first timestep will be set to the baseline concentration, which are the IPCC AR6 1750 values.

```
df_volcanic = pd.read_csv('../tests/test_data/volcanic_ERF_monthly_175001-201912.csv',␣
↪index_col='year')
df_volcanic[1750:].head()
```

```
f.fill_from_rcmip()

# overwrite volcanic
volcanic_forcing = np.zeros(351)
volcanic_forcing[:271] = df_volcanic[1749:].groupby(np.ceil(df_volcanic[1749:].index) //␣
```

(continues on next page)

(continued from previous page)

```
↪1).mean().squeeze().values
fill(f.forcing, volcanic_forcing[:, None, None], specie="Volcanic")  # sometimes need to␣
↪expand the array

initialise(f.concentration, f.species_configs['baseline_concentration'])
initialise(f.forcing, 0)
initialise(f.temperature, 0)
initialise(f.cumulative_emissions, 0)
initialise(f.airborne_emissions, 0)
```

8c. fill climate configs

Take pre-calculated values from the Cummins et al. three layer model. We will use a reproducible random seed to define the stochastic behaviour.

```
df = pd.read_csv("../tests/test_data/4xCO2_cummins_ebm3.csv")
models = df['model'].unique()

seed = 1355763

for config in configs:
    model, run = config.split('_')
    condition = (df['model']==model) & (df['run']==run)
    fill(f.climate_configs['ocean_heat_capacity'], df.loc[condition, 'C1':'C3'].values.
↪squeeze(), config=config)
    fill(f.climate_configs['ocean_heat_transfer'], df.loc[condition, 'kappa1':'kappa3'].
↪values.squeeze(), config=config)
    fill(f.climate_configs['deep_ocean_efficacy'], df.loc[condition, 'epsilon'].
↪values[0], config=config)
    fill(f.climate_configs['gamma_autocorrelation'], df.loc[condition, 'gamma'].
↪values[0], config=config)
    fill(f.climate_configs['sigma_eta'], df.loc[condition, 'sigma_eta'].values[0],␣
↪config=config)
    fill(f.climate_configs['sigma_xi'], df.loc[condition, 'sigma_xi'].values[0],␣
↪config=config)
    fill(f.climate_configs['stochastic_run'], True, config=config)
    fill(f.climate_configs['use_seed'], True, config=config)
    fill(f.climate_configs['seed'], seed, config=config)

    seed = seed + 399
```

9. Run FaIR

look at it go.

You can turn off the progress bar with `progress=False`.

```
f.run()
```

10. Make some nice plots

Presently this is accessed using the `xarray` notation; perhaps we can write a nice filter function like I did with `fill` and `initialise`.

The output attributes of `FAIR` of interest are - temperature (layer=0 is surface) - emissions (an output for GHGs driven with concentration) - concentration (as above, vice versa) - forcing: the per-species effective radiative forcing - forcing_sum: the total forcing - airborne_emissions: total emissions of a GHG remaining in the atmosphere - airborne_fraction: the fraction of GHG emissions remaining in the atmosphere - alpha_lifetime: the scaling factor to unperturbed lifetime. Mutiply the two values to get the atmospheric lifetime of a greenhouse gas (see methane example below) - cumulative_emissions - ocean_heat_content_change - toa_imbalance - stochastic_forcing: if stochastic variability is activated, the non-deterministic part of the forcing

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp119', layer=0)], label=f.
↪configs);
pl.title('ssp119: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
```

```
pl.plot(f.timebounds, f.species_configs['unperturbed_lifetime'].loc[dict(specie='CH4',␣
↪gasbox=0)].data * f.alpha_lifetime.loc[dict(scenario='ssp119', specie='CH4')], label=f.
↪configs);
pl.title('ssp119: methane lifetime')
pl.xlabel('year')
pl.ylabel('methane lifetime (yr)')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='CO2')], label=f.
↪configs);
pl.title('ssp119: CO2 forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='CH4')], label=f.
↪configs);
pl.title('ssp119: methane forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ssp119', specie='CH4')],␣
↪label=f.configs);
pl.title('ssp119: methane concentration')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ssp119', specie='Equivalent␣
↪effective stratospheric chlorine')], label=f.configs);
pl.title('ssp119: EESC')
pl.xlabel('year')
pl.ylabel('ppt')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='N2O')], label=f.
↪configs);
pl.title('ssp119: N2O concentration')
pl.xlabel('year')
pl.ylabel('ppb')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='N2O')], label=f.
↪configs);
pl.title('ssp119: N2O forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ssp119', specie='CH3Cl')],␣
↪label=f.configs);
pl.title('ssp119: Halon-1211 concentration')
pl.xlabel('year')
pl.ylabel('ppt')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Aerosol-radiation␣
↪interactions')], label=f.configs);
pl.title('ssp119: ERFari')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Aerosol-cloud␣
↪interactions')], label=f.configs);
pl.title('ssp119: ERFaci')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Ozone')], label=f.
↪configs);
pl.title('ssp119: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Contrails')],␣
↪label=f.configs);
pl.title('ssp119: Contrails')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Light absorbing␣
↪particles on snow and ice')], label=f.configs);
```

(continues on next page)

(continued from previous page)

```
pl.title('ssp119: LAPSI')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Land use')], label=f.
↪configs);
pl.title('ssp119: land use forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Solar')], label=f.
↪configs);
pl.title('ssp119: solar forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Volcanic')], label=f.
↪configs);
pl.title('ssp119: volcanic forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Stratospheric water␣
↪vapour')], label=f.configs);
pl.title('ssp119: Stratospheric water vapour forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp126', layer=0)], label=f.
↪configs);
pl.title('ssp126: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp245', layer=0)], label=f.
↪configs);
pl.title('ssp245: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp370', layer=0)], label=f.
↪configs);
pl.title('ssp370: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp585', layer=0)], label=f.
↪configs);
pl.title('ssp585: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp126', specie='Ozone')], label=f.
↪configs);
pl.title('ssp126: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp245', specie='Ozone')], label=f.
↪configs);
pl.title('ssp245: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp370', specie='Ozone')], label=f.
↪configs);
pl.title('ssp370: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp585', specie='Ozone')], label=f.
↪configs);
pl.title('ssp585: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.airborne_emissions.loc[dict(scenario='ssp585', specie='CO2')],␣
↪label=f.configs);
pl.title('ssp585: Airborne emissions of CO2')
pl.xlabel('year')
pl.ylabel('GtCO2')
```

```
pl.plot(f.timebounds, f.airborne_fraction.loc[dict(scenario='ssp585', specie='CO2')],␣
↪label=f.configs);
pl.title('ssp585: Airborne fraction of CO2')
pl.xlabel('year')
pl.ylabel('[1]')
```

```
pl.plot(f.timebounds, f.cumulative_emissions.loc[dict(scenario='ssp585', specie='CO2')],␣
↪label=f.configs);
pl.title('ssp585: Cumulative emissions of CO2')
pl.xlabel('year')
pl.ylabel('GtCO2')
```

```
pl.plot(f.timebounds, f.ocean_heat_content_change.loc[dict(scenario='ssp585')], label=f.
↪configs);
```

(continues on next page)

(continued from previous page)

```
pl.title('ssp585: Ocean heat content change')
pl.xlabel('year')
pl.ylabel('J')
```

```
pl.plot(f.timebounds, f.toa_imbalance.loc[dict(scenario='ssp585')], label=f.configs);
pl.title('ssp585: Top of atmosphere energy imbalance')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

```
pl.plot(f.timebounds, f.stochastic_forcing.loc[dict(scenario='ssp585')], label=f.
↪configs);
pl.title('ssp585: Total forcing')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

```
pl.plot(f.timebounds, f.forcing_sum.loc[dict(scenario='ssp585')], label=f.configs);
pl.title('ssp585: Deterministic forcing')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

```
pl.plot(f.timebounds, f.stochastic_forcing.loc[dict(scenario='ssp585')]-f.forcing_sum.
↪loc[dict(scenario='ssp585')], label=f.configs);
pl.title('ssp585: Stochastic forcing component')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

1.3.3 SSP emissions runs using CMIP6 calibrations: monthly timestep

This is based on `cmip6_ssp_emissions_run`. Refer to this example for the full steps; here I take a few shortcuts.

The stochastic response appears to be too small using the calibrated 4xCO2 values for sigma_eta and sigma_xi. No rescaling of these values is done in the energy balance code, though in the constrution of the energy balance model diving the sigmas by the square root of the timestep seems to approximately reproduce the right level of variability. I do this here.

```
import numpy as np
import matplotlib.pyplot as pl
import pandas as pd

from fair import FAIR
from fair.io import read_properties
from fair.interface import fill, initialise
from fair.earth_params import seconds_per_year
```

Set up problem

Create a world running from 1750 to 2100, at monthly intervals

```
f = FAIR(ch4_method='thornhill2021')
```

```
f.define_time(1750, 2100, 1/12)
```

```
f.timebounds[:5], f.timebounds[-5:]
```

```
f.timepoints[:5], f.timepoints[-5:]
```

```
scenarios = ['ssp119', 'ssp126', 'ssp245', 'ssp370', 'ssp434', 'ssp460', 'ssp534-over',
↪'ssp585']
f.define_scenarios(scenarios)
```

```
df = pd.read_csv("../tests/test_data/4xCO2_cummins_ebm3.csv")
models = df['model'].unique()
configs = []

for imodel, model in enumerate(models):
    for run in df.loc[df['model']==model, 'run']:
        configs.append(f"{model}_{run}")
f.define_configs(configs)
```

```
species, properties = read_properties()
```

```
f.define_species(species, properties)
```

```
f.allocate()
```

```
f.emissions
```

```
f.fill_species_configs()
fill(f.species_configs['unperturbed_lifetime'], 10.8537568, specie='CH4')
fill(f.species_configs['baseline_emissions'], 19.01978312, specie='CH4')
fill(f.species_configs['baseline_emissions'], 0.08602230754, specie='N2O')
```

```
df_volcanic = pd.read_csv('../tests/test_data/volcanic_ERF_monthly_175001-201912.csv',␣
↪index_col='year')
```

```
f.fill_from_rcmip()

# overwrite volcanic
volcanic_forcing = np.zeros(4201)
volcanic_forcing[:270*12] = df_volcanic[1750:].squeeze().values
fill(f.forcing, volcanic_forcing[:, None, None], specie="Volcanic")  # sometimes need to␣
↪expand the array

initialise(f.concentration, f.species_configs['baseline_concentration'])
```

(continues on next page)

(continued from previous page)

```
initialise(f.forcing, 0)
initialise(f.temperature, 0)
initialise(f.cumulative_emissions, 0)
initialise(f.airborne_emissions, 0)
```

```
df = pd.read_csv("../tests/test_data/4xCO2_cummins_ebm3.csv")
models = df['model'].unique()

seed = 1355763

# remember we rescale sigma
for config in configs:
    model, run = config.split('_')
    condition = (df['model']==model) & (df['run']==run)
    fill(f.climate_configs['ocean_heat_capacity'], df.loc[condition, 'C1':'C3'].values.
↪squeeze(), config=config)
    fill(f.climate_configs['ocean_heat_transfer'], df.loc[condition, 'kappa1':'kappa3'].
↪values.squeeze(), config=config)
    fill(f.climate_configs['deep_ocean_efficacy'], df.loc[condition, 'epsilon'].
↪values[0], config=config)
    fill(f.climate_configs['gamma_autocorrelation'], df.loc[condition, 'gamma'].
↪values[0], config=config)
    fill(f.climate_configs['sigma_eta'], df.loc[condition, 'sigma_eta'].values[0]/np.
↪sqrt(f.timestep), config=config)
    fill(f.climate_configs['sigma_xi'], df.loc[condition, 'sigma_xi'].values[0]/np.
↪sqrt(f.timestep), config=config)
    fill(f.climate_configs['stochastic_run'], True, config=config)
    fill(f.climate_configs['use_seed'], True, config=config)
    fill(f.climate_configs['seed'], seed, config=config)

    seed = seed + 399
```

Run

```
f.run()
```

Analyse

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp119', layer=0)], label=f.
↪configs);
pl.title('ssp119: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
```

```
import numpy as np
t_mean = np.zeros((350, 66))
for i in range(350):
    t_mean[i, :] = f.temperature.loc[dict(scenario='ssp119', layer=0)][i*12:i*12+12, :].
```

(continues on next page)

(continued from previous page)

```
↪mean(axis=0)

pl.plot(np.arange(1750.5, 2100), t_mean, label=f.configs);
pl.title('ssp119: temperature, annual averages')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
```

```
pl.plot(f.timebounds, f.species_configs['unperturbed_lifetime'].loc[dict(specie='CH4',␣
↪gasbox=0)].data * f.alpha_lifetime.loc[dict(scenario='ssp119', specie='CH4')], label=f.
↪configs);
pl.title('ssp119: methane lifetime')
pl.xlabel('year')
pl.ylabel('methane lifetime (yr)')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='CO2')], label=f.
↪configs);
pl.title('ssp119: CO2 forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='CH4')], label=f.
↪configs);
pl.title('ssp119: methane forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ssp119', specie='CH4')],␣
↪label=f.configs);
pl.title('ssp119: methane concentration')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ssp119', specie='Equivalent␣
↪effective stratospheric chlorine')], label=f.configs);
pl.title('ssp119: EESC')
pl.xlabel('year')
pl.ylabel('ppt')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='N2O')], label=f.
↪configs);
pl.title('ssp119: N2O concentration')
pl.xlabel('year')
pl.ylabel('ppb')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='N2O')], label=f.
↪configs);
pl.title('ssp119: N2O forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.concentration.loc[dict(scenario='ssp119', specie='CH3Cl')],␣
↪label=f.configs);
pl.title('ssp119: Halon-1211 concentration')
pl.xlabel('year')
pl.ylabel('ppt')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Aerosol-radiation␣
↪interactions')], label=f.configs);
pl.title('ssp119: ERFari')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Aerosol-cloud␣
↪interactions')], label=f.configs);
pl.title('ssp119: ERFaci')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Ozone')], label=f.
↪configs);
pl.title('ssp119: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Contrails')],␣
↪label=f.configs);
pl.title('ssp119: Contrails')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Light absorbing␣
↪particles on snow and ice')], label=f.configs);
pl.title('ssp119: LAPSI')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Land use')], label=f.
↪configs);
pl.title('ssp119: land use forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Solar')], label=f.
↪configs);
pl.title('ssp119: solar forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Volcanic')], label=f.
↪configs);
```

(continues on next page)

(continued from previous page)

```
pl.title('ssp119: volcanic forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp119', specie='Stratospheric water␣
↪vapour')], label=f.configs);
pl.title('ssp119: Stratospheric water vapour forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp126', layer=0)], label=f.
↪configs);
pl.title('ssp126: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp245', layer=0)], label=f.
↪configs);
pl.title('ssp245: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp370', layer=0)], label=f.
↪configs);
pl.title('ssp370: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.temperature.loc[dict(scenario='ssp585', layer=0)], label=f.
↪configs);
pl.title('ssp585: temperature')
pl.xlabel('year')
pl.ylabel('Temperature anomaly (K)')
#pl.legend()
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp126', specie='Ozone')], label=f.
↪configs);
pl.title('ssp126: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp245', specie='Ozone')], label=f.
↪configs);
pl.title('ssp245: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp370', specie='Ozone')], label=f.
↪configs);
pl.title('ssp370: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.forcing.loc[dict(scenario='ssp585', specie='Ozone')], label=f.
↪configs);
pl.title('ssp585: Ozone forcing')
pl.xlabel('year')
pl.ylabel('W/m2')
```

```
pl.plot(f.timebounds, f.airborne_emissions.loc[dict(scenario='ssp585', specie='CO2')],␣
↪label=f.configs);
pl.title('ssp585: Airborne emissions of CO2')
pl.xlabel('year')
pl.ylabel('GtCO2')
```

```
pl.plot(f.timebounds, f.airborne_fraction.loc[dict(scenario='ssp585', specie='CO2')],␣
↪label=f.configs);
pl.title('ssp585: Airborne fraction of CO2')
pl.xlabel('year')
pl.ylabel('[1]')
```

```
pl.plot(f.timebounds, f.cumulative_emissions.loc[dict(scenario='ssp585', specie='CO2')],␣
↪label=f.configs);
pl.title('ssp585: Cumulative emissions of CO2')
pl.xlabel('year')
pl.ylabel('GtCO2')
```

```
pl.plot(f.timebounds, f.ocean_heat_content_change.loc[dict(scenario='ssp585')], label=f.
↪configs);
pl.title('ssp585: Ocean heat content change')
pl.xlabel('year')
pl.ylabel('J')
```

```
pl.plot(f.timebounds, f.toa_imbalance.loc[dict(scenario='ssp585')], label=f.configs);
pl.title('ssp585: Top of atmosphere energy imbalance')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

```
pl.plot(f.timebounds, f.stochastic_forcing.loc[dict(scenario='ssp585')], label=f.
↪configs);
pl.title('ssp585: Total forcing')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

```
pl.plot(f.timebounds, f.forcing_sum.loc[dict(scenario='ssp585')], label=f.configs);
pl.title('ssp585: Deterministic forcing')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

```
pl.plot(f.timebounds, f.stochastic_forcing.loc[dict(scenario='ssp585')]-f.forcing_sum.
↪loc[dict(scenario='ssp585')], label=f.configs);
pl.title('ssp585: Stochastic forcing component')
pl.xlabel('year')
pl.ylabel('W m$^{-2}$')
```

1.3.4 CO2 equilibrium runs

Recreate abrupt 4xCO2 runs from all 66 CMIP6 model calibrations and create Gregory plots.

This demonstrates the flexibility of FaIR in that it can be applied to mimic specific ESM experiments.

```
import matplotlib.pyplot as pl
import numpy as np
import pandas as pd

from fair import FAIR
from fair.interface import fill, initialise
```

1. Create FaIR instance

```
f = FAIR()
```

2. Define time horizon

I want 1000 years, even though 4xCO2 is only 150 year experiment.

```
f.define_time(0, 1000, 1)
f.timebounds
```

3. Define scenarios

This is easy: there's only one

```
f.define_scenarios(['abrupt-4xCO2'])
```

4. Define configs

```
df = pd.read_csv("../tests/test_data/4xCO2_cummins_ebm3.csv")
models = df['model'].unique()
configs = []

for imodel, model in enumerate(models):
    for run in df.loc[df['model']==model, 'run']:
        configs.append(f"{model}_{run}")
f.define_configs(configs)
```

5. Define species

Note we set the `input_mode` to forcing, as we are running with prescribed forcing from the 4xCO2 Gregory.

```
species = ['CO2']
```

```
properties = {
    'CO2': {
        'type': 'co2',
        'input_mode': 'forcing',
        'greenhouse_gas': True,
        'aerosol_chemistry_from_emissions': False,
        'aerosol_chemistry_from_concentration': False,
    },
}
```

```
f.define_species(species, properties)
```

6. Modifying run options

Not applicable

7. Create input and output data

```
f.allocate()
```

8. fill in everything

```
initialise(f.temperature, 0)
```

```
df = pd.read_csv("../tests/test_data/4xCO2_cummins_ebm3.csv")
models = df['model'].unique()

seed = 0

for config in configs:
    model, run = config.split('_')
    condition = (df['model']==model) & (df['run']==run)
    fill(f.climate_configs['ocean_heat_capacity'], df.loc[condition, 'C1':'C3'].values.
↪squeeze(), config=config)
    fill(f.climate_configs['ocean_heat_transfer'], df.loc[condition, 'kappa1':'kappa3'].
↪values.squeeze(), config=config)
    fill(f.climate_configs['deep_ocean_efficacy'], df.loc[condition, 'epsilon'].
↪values[0], config=config)
    fill(f.climate_configs['gamma_autocorrelation'], df.loc[condition, 'gamma'].
↪values[0], config=config)
    fill(f.climate_configs['sigma_eta'], df.loc[condition, 'sigma_eta'].values[0],␣
↪config=config)
    fill(f.climate_configs['sigma_xi'], df.loc[condition, 'sigma_xi'].values[0],␣
```

(continues on next page)

(continued from previous page)

```
↪config=config)
    fill(f.climate_configs['stochastic_run'], True, config=config)
    fill(f.climate_configs['use_seed'], True, config=config)
    fill(f.climate_configs['seed'], seed, config=config)

    # We want to fill in a constant 4xCO2 forcing (for each model) across the run.
    fill(f.forcing, df.loc[condition, 'F_4xCO2'].values[0], config=config, specie='CO2')

    seed = seed + 10101
```

```
df
```

```
f.fill_species_configs()
```

```
fill(f.species_configs['tropospheric_adjustment'], 0, specie='CO2')
```

9. run FaIR

```
f.run()
```

10. Show results

Although we can get convincing internal variability for T and N individually, it appears that the stochastic variability is correlated.

```
fig, ax = pl.subplots()
ax.plot(f.timebounds, f.temperature.loc[dict(layer=0, scenario='abrupt-4xCO2')]);
ax.set_xlim(0, 1000)
ax.set_ylim(0, 13)
ax.set_ylabel('Global mean warming above pre-industrial, °C')
ax.set_xlabel('Year')
ax.set_title('CMIP6 abrupt-4xCO$_2$ emulations, FaIR v2.1')
fig.tight_layout()
```

```
pl.plot(f.timebounds, f.toa_imbalance.loc[dict(scenario='abrupt-4xCO2')]);
```

```
pl.plot(f.timebounds, f.forcing_sum.loc[dict(scenario='abrupt-4xCO2')]);
```

```
pl.plot(f.timebounds[800:], f.toa_imbalance.loc[dict(scenario='abrupt-4xCO2')][800:,...])
pl.axhline(0, color='k')
```

```
fig, ax = pl.subplots(11, 6, figsize=(16, 30))

for i, config in enumerate(configs):
    ax[i//6,i%6].scatter(f.temperature.loc[dict(layer=0, scenario='abrupt-4xCO2',␣
↪config=config)], f.toa_imbalance.loc[dict(scenario='abrupt-4xCO2', config=config)])
    ax[i//6,i%6].set_xlim(0,13)
```

(continues on next page)

(continued from previous page)

```
    ax[i//6,i%6].set_ylim(-1, 10)
    ax[i//6,i%6].axhline(0, color='k')
    ax[i//6,i%6].set_title(config, fontsize=6)
```

1.3.5 Run the n-layer energy balance model

This notebook shows examples of extending the 3-layer energy balance model to general n.

For the two and three layer cases we'll take the MLE estimates from Cummins et al. (2020) for HadGEM2-ES, and we'll use the GISS forcing. Where n > 3 the data is fake.

```
import matplotlib.pyplot as pl
import numpy as np
import pandas as pd
import pooch

from fair.energy_balance_model import EnergyBalanceModel
```

```
df_forcing = pd.read_csv('../tests/test_data/RFMIP_ERF_tier2_GISS-E2-1-G.csv')
```

```
ebm3 = EnergyBalanceModel(
    ocean_heat_capacity=[3.62, 9.47, 98.66],
    ocean_heat_transfer=[0.54, 2.39, 0.63],
    deep_ocean_efficacy=1.59,
    gamma_autocorrelation=1.73,
    sigma_xi=0.32,
    sigma_eta=0.43,
    forcing_4co2=6.35,
    stochastic_run=True,
    seed=16
)
```

```
ebm3.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm3.run()
```

```
ebm3.temperature
```

```
time = np.arange(1850.5, 2101)
```

```
pl.plot(time, ebm3.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm3.temperature[:,1], label='second ocean layer')
pl.plot(time, ebm3.temperature[:,2], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
ebm3.emergent_parameters()
ebm3.ecs, ebm3.tcr
```

```
ebm2 = EnergyBalanceModel(
    ocean_heat_capacity=[7.73, 89.29],
    ocean_heat_transfer=[0.63, 0.52],
    deep_ocean_efficacy=1.52,
    gamma_autocorrelation=1.58,
    sigma_xi=0.64,
    sigma_eta=0.43,
    stochastic_run=True,
    forcing_4co2=6.86,
    seed=16
)
```

```
ebm2.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm2.emergent_parameters()
ebm2.ecs, ebm2.tcr
```

```
ebm2.run()
```

```
pl.plot(time, ebm2.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm2.temperature[:,1], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
# this is not based on a tuning to any existing CMIP6 model, but I try to get the TCR␣
↪close to the
# HadGEM2 2- and 3-layer cases.
ebm4 = EnergyBalanceModel(
    ocean_heat_capacity=[1.3, 9, 20, 80],
    ocean_heat_transfer=[0.54, 3, 3, 0.63],
    deep_ocean_efficacy=1.2,
    gamma_autocorrelation=1.73,
    sigma_xi=0.32,
    sigma_eta=0.43,
    forcing_4co2=6.35,
    stochastic_run=True,
    seed=16
)
```

```
ebm4.emergent_parameters()
ebm4.ecs, ebm4.tcr
```

```
ebm4.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm4.run()
```

```
pl.plot(time, ebm4.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm4.temperature[:,1], label='second ocean layer')
pl.plot(time, ebm4.temperature[:,2], label='third ocean layer')
```

(continues on next page)

(continued from previous page)

```
pl.plot(time, ebm4.temperature[:,3], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
# let's go totally crazy
ebm10 = EnergyBalanceModel(
    ocean_heat_capacity=[0.6, 1.3, 2, 5, 7, 10, 45, 70, 80, 130],
    ocean_heat_transfer=[0.54, 4, 5, 5, 5, 5, 5, 5, 5, 0.63],
    deep_ocean_efficacy=1.2,
    gamma_autocorrelation=1.73,
    sigma_xi=0.32,
    sigma_eta=0.43,
    forcing_4co2=6.35,
    stochastic_run=True,
    seed=16
)
```

```
ebm10.emergent_parameters()
ebm10.ecs, ebm10.tcr
```

```
ebm10.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm10.run()
```

```
pl.plot(time, ebm10.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm10.temperature[:,1], label='second ocean layer')
pl.plot(time, ebm10.temperature[:,2], label='third ocean layer')
pl.plot(time, ebm10.temperature[:,3], label='fourth ocean layer')
pl.plot(time, ebm10.temperature[:,4], label='fifth ocean layer')
pl.plot(time, ebm10.temperature[:,5], label='sixth ocean layer')
pl.plot(time, ebm10.temperature[:,6], label='seventh ocean layer')
pl.plot(time, ebm10.temperature[:,7], label='eighth ocean layer')
pl.plot(time, ebm10.temperature[:,8], label='ninth ocean layer')
pl.plot(time, ebm10.temperature[:,9], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
pl.plot(time, ebm2.temperature[:,0], label='two layer model')
pl.plot(time, ebm3.temperature[:,0], label='three layer model')
pl.plot(time, ebm4.temperature[:,0], label='four layer model')
pl.plot(time, ebm10.temperature[:,0], label='ten layer model')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
pl.plot(time, ebm2.toa_imbalance, label='two layer model')
pl.plot(time, ebm3.toa_imbalance, label='three layer model')
pl.plot(time, ebm4.toa_imbalance, label='four layer model')
```

(continues on next page)

(continued from previous page)

```
pl.plot(time, ebm10.toa_imbalance, label='ten layer model')
pl.ylabel('W/m2 relative to 1850')
pl.title('SSP2-4.5 TOA radiation change')
pl.legend()
```

Repeat everything with stochastic forcing switched off

```
ebm3 = EnergyBalanceModel(
    ocean_heat_capacity=[3.62, 9.47, 98.66],
    ocean_heat_transfer=[0.54, 2.39, 0.63],
    deep_ocean_efficacy=1.59,
    gamma_autocorrelation=1.73,
    sigma_xi=0.32,
    sigma_eta=0.43,
    forcing_4co2=6.35,
    stochastic_run=False,
    seed=16
)
```

```
ebm3.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm3.run()
```

```
ebm3.temperature
```

```
time = np.arange(1850.5, 2101)
```

```
pl.plot(time, ebm3.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm3.temperature[:,1], label='second ocean layer')
pl.plot(time, ebm3.temperature[:,2], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
ebm3.emergent_parameters()
ebm3.ecs, ebm3.tcr
```

```
ebm2 = EnergyBalanceModel(
    ocean_heat_capacity=[7.73, 89.29],
    ocean_heat_transfer=[0.63, 0.52],
    deep_ocean_efficacy=1.52,
    gamma_autocorrelation=1.58,
    sigma_xi=0.64,
    sigma_eta=0.43,
    stochastic_run=False,
    forcing_4co2=6.86,
    seed=16
)
```

```
ebm2.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm2.emergent_parameters()
ebm2.ecs, ebm2.tcr
```

```
ebm2.run()
```

```
pl.plot(time, ebm2.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm2.temperature[:,1], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
# this is not based on a tuning to any existing CMIP6 model, but I try to get the TCR␣
↪close to the
# HadGEM2 2- and 3-layer cases.
ebm4 = EnergyBalanceModel(
    ocean_heat_capacity=[1.3, 9, 20, 80],
    ocean_heat_transfer=[0.54, 3, 3, 0.63],
    deep_ocean_efficacy=1.2,
    gamma_autocorrelation=1.73,
    sigma_xi=0.32,
    sigma_eta=0.43,
    forcing_4co2=6.35,
    stochastic_run=False,
    seed=16
)
```

```
ebm4.emergent_parameters()
ebm4.ecs, ebm4.tcr
```

```
ebm4.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm4.run()
```

```
pl.plot(time, ebm4.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm4.temperature[:,1], label='second ocean layer')
pl.plot(time, ebm4.temperature[:,2], label='third ocean layer')
pl.plot(time, ebm4.temperature[:,3], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
# let's go totally crazy
ebm10 = EnergyBalanceModel(
    ocean_heat_capacity=[0.6, 1.3, 2, 5, 7, 10, 45, 70, 80, 130],
    ocean_heat_transfer=[0.54, 4, 5, 5, 5, 5, 5, 5, 5, 0.63],
    deep_ocean_efficacy=1.2,
    gamma_autocorrelation=1.73,
    sigma_xi=0.32,
```

(continues on next page)

(continued from previous page)

```
    sigma_eta=0.43,
    forcing_4co2=6.35,
    stochastic_run=False,
    seed=16
)
```

```
ebm10.emergent_parameters()
ebm10.ecs, ebm10.tcr
```

```
ebm10.add_forcing(forcing = df_forcing['GISS-E2-1-G TOT'].values, timestep=1)
```

```
ebm10.run()
```

```
pl.plot(time, ebm10.temperature[:,0], label='surface / top ocean layer')
pl.plot(time, ebm10.temperature[:,1], label='second ocean layer')
pl.plot(time, ebm10.temperature[:,2], label='third ocean layer')
pl.plot(time, ebm10.temperature[:,3], label='fourth ocean layer')
pl.plot(time, ebm10.temperature[:,4], label='fifth ocean layer')
pl.plot(time, ebm10.temperature[:,5], label='sixth ocean layer')
pl.plot(time, ebm10.temperature[:,6], label='seventh ocean layer')
pl.plot(time, ebm10.temperature[:,7], label='eighth ocean layer')
pl.plot(time, ebm10.temperature[:,8], label='ninth ocean layer')
pl.plot(time, ebm10.temperature[:,9], label='deep ocean layer')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
pl.plot(time, ebm2.temperature[:,0], label='two layer model')
pl.plot(time, ebm3.temperature[:,0], label='three layer model')
pl.plot(time, ebm4.temperature[:,0], label='four layer model')
pl.plot(time, ebm10.temperature[:,0], label='ten layer model')
pl.ylabel('K relative to 1850')
pl.title('SSP2-4.5 temperature change')
pl.legend()
```

```
pl.plot(time, ebm2.toa_imbalance, label='two layer model')
pl.plot(time, ebm3.toa_imbalance, label='three layer model')
pl.plot(time, ebm4.toa_imbalance, label='four layer model')
pl.plot(time, ebm10.toa_imbalance, label='ten layer model')
pl.ylabel('W/m2 relative to 1850')
pl.title('SSP2-4.5 TOA radiation change')
pl.legend()
```

1.4 API reference

1.4.1 fair.fair

Initialiser module for FaIR.

class fair.**FAIR**(*n_gasboxes=4*, *n_layers=3*, *iirf_max=100*, *br_cl_ods_potential=45*, *ghg_method='meinshausen2020'*, *ch4_method='leach2021'*, *temperature_prescribed=False*)

Initialise FaIR.

Parameters

- **n_gasboxes** (*int*) – the number of atmospheric greenhouse gas boxes to run the model with
- **n_layers** (*int*) – the number of ocean layers in the energy balance or impulse response model to run with
- **iirf_max** (*float*) – limit for time-integral of greenhouse gas impulse response function.
- **br_cl_ods_potential** (*float*) – factor describing the ratio of efficiency that each bromine atom has as an ozone depleting substance relative to each chlorine atom.
- **ghg_method** (*str*) – method to use for calculating greenhouse gas forcing from CO_2, CH_4 and N:sub:`2`O. Valid options are {"leach2021", "meinshausen2020", "etminan2016", "myhre1998"}
- **ch4_method** (*str*) – method to use for calculating methane lifetime change. Valid options are {"leach2021", "thornhill2021"}.
- **temperature_prescribed** (*bool*) – Run FaIR with temperatures prescribed.

Raises

ValueError – if ghg_method or ch4_method given are not valid options.

allocate()

Create xarray DataArrays of data input and output.

calculate_concentration_per_emission(*mass_atmosphere=5.1352e+18*, *molecular_weight_air=28.97*)

Calculate change in atmospheric concentration for each unit emission.

calculate_g(*iirf_horizon=100*)

Calculate lifetime scaling parameters.

Parameters

iirf_horizon (*float*) – time horizon for time-integrated airborne fraction (yr).

calculate_iirf0(*iirf_horizon=100*)

Convert greenhouse gas lifetime to time-integrated airborne fraction.

iirf_0 is the 100-year time-integrated airborne fraction to a pulse emission. We know that the gas's atmospheric airborne fraction $a(t)$ for a gas with lifetime τ after time t is therefore

$$a(t) = \exp(-t/tau)$$

and integrating this for H years after a pulse emissions gives us:

$$r_0(t) = \int_0^H \exp(-t/\tau)dt = \tau(1 - \exp(-H/\tau)).$$

H = 100 years is the default time horizon in FaIR but this can be set to any value.

Parameters
iirf_horizon (*float*) – time horizon for time-integrated airborne fraction (yr).

property ch4_method

Return methane lifetime method.

define_configs(*configs*)

Define configs to analyse in FaIR.

Parameters
configs (*list*) – config names to run

define_scenarios(*scenarios*)

Define scenarios to analyse in FaIR.

Parameters
scenarios (*list*) – scenario names to run

define_species(*species*, *properties*)

Define species to run in FaIR.

Parameters

- **species** (*list*) – names of species to include in FaIR
- **properties** (*dict*) – mapping of each specie to particular run properties. This is a nested dict, where each dict key contains a dict of 5 keys as follows:

  type
  [str] the type of specie that is being provided. Valid inputs are "co2 ffi", "co2 afolu", "co2", "ch4", "n2o", "cfc-11", "other halogen", "f-gas", "sulfur", "black carbon", "organic carbon", "other slcf", "nox aviation", "eesc", "ozone", "ari", "aci", "contrails", "lapsi","h2o stratospheric", "land use", "volcanic", "solar", "unspecified",

  input_mode
  [{'emissions', 'concentration', 'forcing', 'calculated'}] describes how the specie is input into the model.

  greenhouse_gas
  [bool] is the specie a greenhouse gas?

  aerosol_chemistry_from_emissions
  [bool] does the specie's emissions affect aerosols and/or chemistry?

  aerosol_chemistry_from_concentration
  [bool] does the specie's concentration affect aerosols and/or chemistry?

Raises

- **ValueError** – if a specie in species does not have a matching key in properties.
- **ValueError** – if an invalid species type is specified.
- **ValueError** – if an invalid input_type (driving mode) is provided for a particular type.
- **ValueError** – if duplicate species types are provided for types that must be unique.

define_time(*start*, *end*, *step*)

Define timebounds vector to run FaIR.

Parameters

- **start** (*float*) – first timebound of the model (year)

- **end** (*float*) – last timebound of the model (year)
- **step** (*float*) – timestep (year)

fill_from_rcmip()

Fill emissions, concentrations and/or forcing from RCMIP scenarios.

fill_species_configs(*filename='/home/docs/checkouts/readthedocs.org/user_builds/fair/envs/v2.1.0/lib/python3.6/site-packages/fair/defaults/data/ar6/species_configs_properties.csv'*)

Fill the species_configs with values from a CSV file.

Parameters

filename (*str*) – Path of the CSV file to read the species configs from. If omitted, the default configs file will be read.

property ghg_method

Return greenhouse gas forcing method.

run(*progress=True*, *suppress_warnings=True*)

Run the FaIR model.

Parameters

- **progress** (*bool*) – Display progress bar.
- **suppress_warnings** (*bool*) – Hide warnings relating to covariance in energy balance matrix.

to_netcdf(*filename*)

Write out FaIR scenario data to a netCDF file.

Parameters

filename (*str*) – file path of the file to write.

1.4.2 fair.energy_balance_model

n-layer energy balance representation of Earth's climate.

class fair.energy_balance_model.**EnergyBalanceModel**(*ocean_heat_capacity*, *ocean_heat_transfer*, *deep_ocean_efficacy=1*, *forcing_4co2=8*, *stochastic_run=False*, *sigma_eta=0.5*, *sigma_xi=0.5*, *gamma_autocorrelation=2*, *seed=None*, *timestep=1*, *n_timesteps=1*)

Energy balance model that converts forcing to temperature.

The energy balance model is converted to an impulse-response formulation (hence the IR part of FaIR) to allow efficient evaluation. The benefits of this are increased as once derived, the "layers" of the energy balance model do not communicate with each other. The model description can be found in [Leach2021], [Cummins2020], [Tsutsui2017] and [Geoffroy2013].

Parameters

- **ocean_heat_capacity** (np.ndarray) – Ocean heat capacity of each layer (top first), W m-2 yr K-1
- **ocean_heat_transfer** (np.ndarray) – Heat exchange coefficient between ocean layers (top first). The first element of this array is akin to the climate feedback parameter, with the convention that stabilising feedbacks are positive (opposite to most climate sensitivity literature). W m-2 K-1

- **deep_ocean_efficacy** (*float*) – efficacy of deepest ocean layer. See e.g. [Geoffroy2013].
- **forcing_4co2** (*float*) – effective radiative forcing from a quadrupling of atmospheric CO2 concentrations above pre-industrial.
- **stochastic_run** (*bool*) – Activate the stochastic variability component from [Cummins2020].
- **sigma_eta** (*float*) – Standard deviation of stochastic forcing component from [Cummins2020].
- **sigma_xi** (*float*) – Standard deviation of stochastic disturbance applied to surface layer. See [Cummins2020].
- **gamma_autocorrelation** (*float*) – Stochastic forcing continuous-time autocorrelation parameter. See [Cummins2020].
- **seed** (*int or None*) – Random seed to use for stochastic variability.
- **timestep** (*float*) – Time interval of the model (yr)

Raises

- **ValueError** – if the shapes of `ocean_heat_capacity` and `ocean_heat_transfer` differ.
- **ValueError** – if there are not at least two layers in the energy balance model.

add_forcing(*forcing*, *timestep*)

Add a forcing time series to EnergyBalanceModel.

Parameters

- **forcing** (*np.ndarray*) – time series of [effective] radiative forcing
- **timestep** (*float*) – Model timestep, in years

property eb_matrix_d

Return the discretised matrix exponential.

emergent_parameters(*forcing_2co2_4co2_ratio=0.5*)

Calculate emergent parameters from the energy balance parameters.

Parameters

forcing_2co2_4co2_ratio (*float*) – ratio of (effective) radiative forcing converting a quadrupling of CO2 to a doubling of CO2.

property forcing_vector_d

Return the discretised forcing vector.

impulse_response()

Convert the energy balance to impulse response representation.

run()

Run the EnergyBalanceModel.

property stochastic_d

Return the stochastic matrix.

fair.energy_balance_model.**calculate_toa_imbalance_postrun**(*state*, *forcing*, *ocean_heat_transfer*, *deep_ocean_efficacy*)

Calculate top of atmosphere energy imbalance.

The calculation is performed after the scenario has been run to avoid looping, since no dynamic state changes affect the calculation.

Parameters

- **state** (*np.ndarray*) – stacked arrays of forcing and temperature of layers across the run
- **forcing** (*np.ndarray*) – stacked arrays of [effective] radiative forcing across the run
- **ocean_heat_transfer** (*np.ndarray*) – Heat exchange coefficient between ocean layers (top first). The first element of this array is akin to the climate feedback parameter, with the convention that stabilising feedbacks are positive (opposite to most climate sensitivity literature). W m-2 K-1
- **deep_ocean_efficacy** (*np.ndarray*) – efficacy of deepest ocean layer.

Returns

toa_imbalance – Top of atmsophere energy imbalance.

Return type

np.ndarray

fair.energy_balance_model.**multi_ebm**(*configs*, *ocean_heat_capacity*, *ocean_heat_transfer*, *deep_ocean_efficacy*, *stochastic_run*, *sigma_eta*, *sigma_xi*, *gamma_autocorrelation*, *seed*, *use_seed*, *forcing_4co2*, *timestep*, *timebounds*)

Create several instances of the EnergyBalanceModel.

This allows efficient parallel implementation in FaIR. We have to use a for loop in this function as is does not look like the linear algebra functions in scipy are naturally parallel.

Parameters

- **configs** (*list*) – A named list of climate configurations.
- **ocean_heat_capacity** (*np.ndarray*) – Ocean heat capacity of each layer (top first), W m-2 yr K-1
- **ocean_heat_transfer** (*np.ndarray*) – Heat exchange coefficient between ocean layers (top first). The first element of this array is akin to the climate feedback parameter, with the convention that stabilising feedbacks are positive (opposite to most climate sensitivity literature). W m-2 K-1
- **deep_ocean_efficacy** (*float*) – efficacy of deepest ocean layer. See e.g. [Geoffroy2013].
- **stochastic_run** (*bool*) – Activate the stochastic variability component from [Cummins2020].
- **sigma_eta** (*float*) – Standard deviation of stochastic forcing component from [Cummins2020].
- **sigma_xi** (*float*) – Standard deviation of stochastic disturbance applied to surface layer. See [Cummins2020].
- **gamma_autocorrelation** (*float*) – Stochastic forcing continuous-time autocorrelation parameter. See [Cummins2020].
- **seed** (*int or None*) – Random seed to use for stochastic variability.
- **use_seed** (*bool*) – Whether or not to use the random seed.
- **forcing_4co2** (*float*) – effective radiative forcing from a quadrupling of atmospheric CO2 concentrations above pre-industrial.
- **timestep** (*float*) – Time interval of the model (yr)
- **timebounds** (*np.ndarray*) – Vector representing the time snapshots to calculate temperature on.

fair.energy_balance_model.**step_temperature**(*state_old*, *eb_matrix_d*, *forcing_vector_d*, *stochastic_d*, *forcing*)

Advance parallel energy balance models forward one timestep.

Parameters

- **state_old** (*np.ndarray*) – stacked arrays of forcing and temperature of layers in previous timestep
- **eb_matrix_d** (*np.ndarray*) – stacked discretised energy balance matrices
- **forcing_vector_d** (*np.ndarray*) – stacked discretised forcing vectors
- **_stochastic_d** (*np.ndarray*) – stacked matrices of stochastic internal variability
- **forcing** (*np.ndarray*) – stacked vectors of [effective] radiative forcing

Returns

state_new – stacked arrays of forcing and temperature of layers in this timestep

Return type

np.ndarray

1.4.3 fair.interface

Convenience functions for filling FaIR *xarray* instances.

fair.interface.**fill**(*var*, *data*, ***kwargs*)

Fill a FAIR variable instance.

One could work directly with the xarray DataArrays, but this function includes additional error checking and validation, and is slightly less cumbersome than the xarray method of allocation.

Parameters

- **var** (*attr*) – FAIR variable attribute to fill, for example fair.climate_configs["ocean_heat_capacity"]
- **data** (*np.ndarray*) – data to fill the variable with
- ****kwargs** – the dimensions represeted in data

Raises

ValueError – if a kwargs element provided doesn't correspond to a dimension name in var

fair.interface.**initialise**(*var*, *value*, ***kwargs*)

Fill a fair variable instance with value in first timebound.

Otherwise identical to fill.

Parameters

- **var** (*attr*) – FAIR variable attribute to fill for example fair.climate_configs["ocean_heat_capacity"]
- **value** (*np.ndarray*) – value to fill the first timebound with
- ****kwargs** – the dimensions represeted in data

1.4.4 fair.io

Tools for getting data into and out of FaIR.

fair.io.**read_properties**(*filename='/home/docs/checkouts/readthedocs.org/user_builds/fair/envs/v2.1.0/lib/python3.6/site-packages/fair/defaults/data/ar6/species_configs_properties.csv'*, *species=None*)

Get a properties file.

Parameters

- **filename** (*str*) – path to a csv file. Default is an AR6 WG1-like config for FaIR covering all of the species considered in CMIP6.
- **species** (*list of str or None*) – the species that are to be included in the FaIR run. All of these species should be present in the index (first column) of the csv. If None (default), return all of the species in the defaults.

Returns

- **species** (*list*) – a list of species names that are included in the FaIR run.
- **properties** (*dict*) – species properties that control the FaIR run

1.4.5 fair.constants

General constants.

fair.constants.**DOUBLING_TIME_1PCT = 69.66071689357483**

Time in years for a doubling to occur at a rate of 1% per year.

fair.constants.**TIME_AXIS = 0**

Axis of the data arrays referencing time.

fair.constants.**SCENARIO_AXIS = 1**

Axis of the data arrays referencing scenario.

fair.constants.**CONFIG_AXIS = 2**

Axis of the data arrays referencing climate or specie configuration.

fair.constants.**SPECIES_AXIS = 3**

Axis of emissions, concentration and forcing data arrays representing species.

fair.constants.**GASBOX_AXIS = 4**

Axis of atmopsheric gas box for the gas partition data array in full emissions to concentration mode.

1.4.6 fair.earth_params

Parameters (not constants) that define Earth properties.

fair.earth_params.**molecular_weight_air = 28.97**

Molecular weight of air, g/mol.

fair.earth_params.**earth_radius = 6371000**

Radius of Earth, m

fair.earth_params.**mass_atmosphere = 5.1352e+18**

Mass of Earth's atmosphere, kg.

fair.earth_params.**seconds_per_year = 31556925.216**

Length of a tropical year, s.

1.4.7 fair.forcing

Module for effective radiative forcing.

fair.forcing.ghg

Module for greenhouse gas forcing.

fair.forcing.ghg.**etminan2016**(*concentration*, *baseline_concentration*, *forcing_scaling*, *radiative_efficiency*, *co2_indices*, *ch4_indices*, *n2o_indices*, *minor_greenhouse_gas_indices*, *a1=-2.4e-07*, *b1=0.00072*, *c1=-0.00021*, *d1=5.36*, *a2=-8e-06*, *b2=4.2e-06*, *c2=-4.9e-06*, *d2=0.117*, *a3=-1.3e-06*, *b3=-8.2e-06*, *d3=0.043*)

Greenhouse gas forcing from concentrations.

This uses the [Etminan2016] relationships for CO2, CH4 and N2O including band overlaps between these three gases. Forcing from minor greenhouse gases are a linear function of their concentration based on their radiative efficiency.

Note that the relationship can be poorly behaved outside of the range of valid values in [Etminan2016], (180-2000 ppm CO2, 340-3500 ppb CH4, 200-525 ppb N2O) particularly for CO2 concentrations above 2000 ppm. It is recommended to use "meinshausen2020", which is a re-fit of the coefficients and extension to be better behaved outside of the valid concentration range.

Parameters

- **concentration** (*np.ndarray*) – concentration of greenhouse gases. "CO2", "CH4" and "N2O" must be included in units of [ppm, ppb, ppb]. Other GHGs are units of ppt.
- **baseline_concentration** (*np.ndarray*) – pre-industrial concentration of the gases (see above).
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **radiative_efficiency** (*np.ndarray*) – radiative efficiency to use for linear-forcing gases, in W m-2 ppb-1
- **co2_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CO2.
- **ch4_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CH4.
- **n2o_indices** (*np.ndarray of bool*) – index along SPECIES AXIS relating to N2O.
- **minor_greenhouse_gas_indices** (*np.ndarray of bool*) – indices of other GHGs that are not CO2, CH4 or N2O.
- **a1** (*float*) – fitting parameter (see [Etminan2016])
- **b1** (*float*) – fitting parameter (see [Etminan2016])
- **c1** (*float*) – fitting parameter (see [Etminan2016])
- **d1** (*float*) – fitting parameter (see [Etminan2016])
- **a2** (*float*) – fitting parameter (see [Etminan2016])

- **b2** (*float*) – fitting parameter (see [Etminan2016])
- **c2** (*float*) – fitting parameter (see [Etminan2016])
- **d2** (*float*) – fitting parameter (see [Etminan2016])
- **a3** (*float*) – fitting parameter (see [Etminan2016])
- **b3** (*float*) – fitting parameter (see [Etminan2016])
- **d3** (*float*) – fitting parameter (see [Etminan2016])

Returns
effective_radiative_forcing – effective radiative forcing (W/m2) from greenhouse gases

Return type
np.ndarray

fair.forcing.ghg.**leach2021ghg**(*concentration*, *baseline_concentration*, *forcing_scaling*, *radiative_efficiency*, *co2_indices*, *ch4_indices*, *n2o_indices*, *minor_greenhouse_gas_indices*, *f1_co2=4.57*, *f3_co2=0.086*, *f3_ch4=0.038*, *f3_n2o=0.106*)

Greenhouse gas forcing from concentrations.

This uses the [Leach2021] relationships for CO2, CH4 and N2O that do not include band overlaps between these gases, allowing single-forcing runs. This is the default treatment in FaIR2.0. This is a re-fit of the [Etminan2016] formulation with improved coefficient fits. Minor greenhouse gases are a linear function of their concentration based on their radiative efficiency.

Parameters

- **concentration** (*np.ndarray*) – concentration of greenhouse gases. "CO2", "CH4" and "N2O" must be included in units of [ppm, ppb, ppb]. Other GHGs are units of ppt.
- **baseline_concentration** (*np.ndarray*) – pre-industrial concentration of the gases (see above).
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **radiative_efficiency** (*np.ndarray*) – radiative efficiency to use for linear-forcing gases, in W m-2 ppb-1
- **co2_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CO2.
- **ch4_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CH4.
- **n2o_indices** (*np.ndarray of bool*) – index along SPECIES AXIS relating to N2O.
- **minor_greenhouse_gas_indices** (*np.ndarray of bool*) – indices of other GHGs that are not CO2, CH4 or N2O.
- **f1_co2** (*float*) – factor relating logarithm of CO2 conentration to radiative forcing.
- **f3_co2** (*float*) – factor relating square root of CO2 conentration to radiative forcing.
- **f3_ch4** (*float*) – factor relating square root of CH4 conentration to radiative forcing.
- **f3_n2o** (*float*) – factor relating square root of N2O conentration to radiative forcing.

Returns
effective_radiative_forcing – effective radiative forcing (W/m2) from greenhouse gases

Return type
np.ndarray

fair.forcing.ghg.**meinshausen2020**(*concentration*, *reference_concentration*, *forcing_scaling*, *radiative_efficiency*, *co2_indices*, *ch4_indices*, *n2o_indices*, *minor_greenhouse_gas_indices*, *a1=-2.4785e-07*, *b1=0.00075906*, *c1=-0.0021492*, *d1=5.2488*, *a2=-0.00034197*, *b2=0.00025455*, *c2=-0.00024357*, *d2=0.12173*, *a3=-8.9603e-05*, *b3=-0.00012462*, *d3=0.045194*)

Greenhouse gas forcing from concentrations.

This uses the [Meinshausen2020] relationships for CO2, CH4 and N2O including band overlaps between these three gases. This is a rescaled [Etminan2016] function with improved stability outside the range of validity in [Etminan2016]. Minor greenhouse gases are a linear function of their concentration based on their radiative efficiency.

Parameters

- **concentration** (*np.ndarray*) – concentration of greenhouse gases. "CO2", "CH4" and "N2O" must be included in units of [ppm, ppb, ppb]. Other GHGs are units of ppt.
- **reference_concentration** (*np.ndarray*) – pre-industrial concentration of the gases (see above) used as the reference to calculate the forcing.
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **radiative_efficiency** (*np.ndarray*) – radiative efficiency to use for linear-forcing gases, in W m-2 ppb-1
- **co2_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CO2.
- **ch4_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CH4.
- **n2o_indices** (*np.ndarray of bool*) – index along SPECIES AXIS relating to N2O.
- **minor_greenhouse_gas_indices** (*np.ndarray of bool*) – indices of other GHGs that are not CO2, CH4 or N2O.
- **a1** (*float*) – fitting parameter (see [Meinshausen2020])
- **b1** (*float*) – fitting parameter (see [Meinshausen2020])
- **c1** (*float*) – fitting parameter (see [Meinshausen2020])
- **d1** (*float*) – fitting parameter (see [Meinshausen2020])
- **a2** (*float*) – fitting parameter (see [Meinshausen2020])
- **b2** (*float*) – fitting parameter (see [Meinshausen2020])
- **c2** (*float*) – fitting parameter (see [Meinshausen2020])
- **d2** (*float*) – fitting parameter (see [Meinshausen2020])
- **a3** (*float*) – fitting parameter (see [Meinshausen2020])
- **b3** (*float*) – fitting parameter (see [Meinshausen2020])
- **d3** (*float*) – fitting parameter (see [Meinshausen2020])

Returns

effective_radiative_forcing – effective radiative forcing (W/m2) from greenhouse gases

Return type

np.ndarray

fair.forcing.ghg.**myhre1998**(*concentration*, *baseline_concentration*, *forcing_scaling*, *radiative_efficiency*, *co2_indices*, *ch4_indices*, *n2o_indices*, *minor_greenhouse_gas_indices*, *alpha_co2=5.35*, *alpha_ch4=0.036*, *alpha_n2o=0.12*, *alpha_ch4_n2o=0.47*, *a1=2.01e-05*, *exp1=0.75*, *a2=5.32e-15*, *exp2=1.52*)

Greenhouse gas forcing from concentrations.

Band overlaps are included between CH4 and N2O. This relationship comes from [Myhre1998], and was used up until the IPCC's Fifth Assessment Report. Minor greenhouse gases are a linear function of their concentration based on their radiative efficiency.

Parameters

- **concentration** (*np.ndarray*) – concentration of greenhouse gases. "CO2", "CH4" and "N2O" must be included in units of [ppm, ppb, ppb]. Other GHGs are units of ppt.
- **baseline_concentration** (*np.ndarray*) – pre-industrial concentration of the gases (see above).
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **radiative_efficiency** (*np.ndarray*) – radiative efficiency to use for linear-forcing gases, in W m-2 ppb-1
- **co2_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CO2.
- **ch4_indices** (*np.ndarray of bool*) – index along SPECIES_AXIS relating to CH4.
- **n2o_indices** (*np.ndarray of bool*) – index along SPECIES AXIS relating to N2O.
- **minor_greenhouse_gas_indices** (*np.ndarray of bool*) – indices of other GHGs that are not CO2, CH4 or N2O.
- **alpha_co2** (*float*) – factor relating logarithm of CO2 conentration to radiative forcing.
- **alpha_ch4** (*float*) – factor relating square root of CH4 conentration to radiative forcing.
- **alpha_n2o** (*float*) – factor relating square root of N2O conentration to radiative forcing.

Returns
effective_radiative_forcing – effective radiative forcing (W/m2) from greenhouse gases

Return type
np.ndarray

fair.forcing.minor

Module for a generic linear emissions or concentration to forcing calculation.

fair.forcing.minor.**calculate_linear_forcing**(*driver*, *baseline_driver*, *forcing_scaling*, *radiative_efficiency*)

Calculate effective radiative forcing from a linear relationship.

Parameters

- **driver** (*np.ndarray*) – input emissions, concentration or forcing
- **baseline_driver** (*np.ndarray*) – baseline, possibly pre-industrial, emissions, concentration or forcing.
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).

- **radiative_efficiency** (*np.ndarray*) – radiative efficiency (W m^{-2} (<driver unit>)$^{-1}$) of each species.

Returns
erf_out – effective radiative forcing (W m^{-2})

Return type
np.ndarray

fair.forcing.ozone

Module for ozone forcing.

`fair.forcing.ozone.`**thornhill2021**(*emissions*, *concentration*, *baseline_emissions*, *baseline_concentration*, *forcing_scaling*, *ozone_radiative_efficiency*, *slcf_indices*, *ghg_indices*)

Determine ozone effective radiative forcing.

Calculates total ozone forcing from precursor emissions and concentrations based on AerChemMIP and CMIP6 Historical behaviour in [Skeie2020] and [Thornhill2021a].

The treatment is identical to FaIR2.0 [Leach2021].

Parameters

- **emissions** (*np.ndarray*) – emissions in timestep
- **concentration** (*np.ndarray*) – concentrations in timestep
- **baseline_emissions** (*np.ndarray*) – reference, possibly pre-industrial, emissions
- **baseline_concentration** (*np.ndarray*) – reference, possibly pre-industrial, concentrations
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **ozone_radiative_efficiency** (*np.ndarray*) – the radiative efficiency at which ozone precursor emissions or concentrations are converted to ozone radiative forcing. The unit is W m-2 (<native emissions or concentration unit>)-1, where the emissions unit is usually Mt/yr for short-lived forcers, ppb for CH4 and N2O concentrations, and ppt for halogenated species. Note this is not the same radiative efficiency that is used in converting GHG concentrations to forcing.
- **slcf_indices** (*list of int*) – provides a mapping of which aerosol species corresponds to which emitted species index along the SPECIES_AXIS.
- **ghg_indices** (*list of int*) – provides a mapping of which aerosol species corresponds to which atmospheric GHG concentration along the SPECIES_AXIS.

Returns
_erf_out – ozone forcing

Return type
np.ndarray

fair.forcing.aerosol

Module for aerosol effective radiative forcing.

fair.forcing.aerosol.erfari

Module for calculating forcing from aerosol-radiation interactions.

fair.forcing.aerosol.erfari.**calculate_erfari_forcing**(*emissions*, *concentration*, *baseline_emissions*, *baseline_concentration*, *forcing_scaling*, *radiative_efficiency*, *emissions_indices*, *concentration_indices*)

Calculate effective radiative forcing from aerosol-radiation interactions.

Parameters

- **emissions** (*np.ndarray*) – emissions in timestep
- **concentration** (*np.ndarray*) – concentrations in timestep
- **baseline_emissions** (*np.ndarray*) – baseline emissions to evaluate forcing against
- **baseline_concentration** (*np.ndarray*) – baseline concentrations to evaluate forcing against
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **radiative_efficiency** (*np.ndarray*) – radiative efficiency (W m-2 (emission_unit yr-1)-1) of each species.
- **emissions_indices** (*list of int*) – provides a mapping of which aerosol species corresponds to which emitted species index along the SPECIES_AXIS.
- **concentration_indices** (*list of int*) – provides a mapping of which aerosol species corresponds to which atmospheric GHG concentration along the SPECIES_AXIS.

Returns

effective_radiative_forcing – effective radiative forcing (W m^{-2}) from aerosol-radiation interactions

Return type

np.ndarray

fair.forcing.aerosol.erfaci

Module for forcing from aerosol-cloud interactions.

fair.forcing.aerosol.erfaci.**logsum**(*emissions*, *concentration*, *baseline_emissions*, *baseline_concentration*, *forcing_scaling*, *scale*, *sensitivity*, *slcf_indices*, *ghg_indices*)

Calculate effective radiative forcing from aerosol-cloud interactions.

This uses the relationship to calculate ERFaci as follows

$$F = \beta \log\left(1 + \sum_i s_i A_i\right)$$

where A_i is the emissions or concentration of a specie, β is the scale factor and s_i describes how sensitive a specie is in contributing to ERFaci.

The calculation is performed for the emissions/concentration of interest and then for the baseline. The difference between the two values is the forcing.

This relationship is a generalisation of [Stevens2015]. To recover [Stevens2015], set s_i to zero for all except SO_2, and $s_{SO2} = 1/Q_n$ where Q_n is the natural emissions source in [Stevens2015].

Parameters

- **emissions** (*np.ndarray*) – input emissions
- **concentration** (*np.ndarray*) – input emissions
- **baseline_emissions** (*np.ndarray*) – baseline emissions
- **baseline_concentration** (*np.ndarray*) – baseline concentration
- **forcing_scaling** (*np.ndarray*) – scaling of the calculated radiative forcing (e.g. for conversion to effective radiative forcing and forcing uncertainty).
- **scale** (*np.ndarray*) – per-species scaling factor to apply to the logarithm
- **sensitivity** (*np.ndarray*) – per-species sensitivity factor for the logarithm
- **slcf_indices** (*list of int*) – provides a mapping of which aerosol species corresponds to which emitted species index along the SPECIES_AXIS.
- **ghg_indices** (*list of int*) – provides a mapping of which aerosol species corresponds to which atmospheric GHG concentration along the SPECIES_AXIS.

Returns

effective_radiative_forcing – effective radiative forcing (W m^{-2}) from aerosol-cloud interactions

Return type

np.ndarray

1.4.8 fair.gas_cycle

Module containing gas cycle functions.

fair.gas_cycle.ch4_lifetime

Methane lifetime definition that is based on multiple species.

fair.gas_cycle.ch4_lifetime.**calculate_alpha_ch4**(*emissions*, *concentration*, *temperature*, *baseline_emissions*, *baseline_concentration*, *ch4_lifetime_chemical_sensitivity*, *ch4_lifetime_temperature_sensitivity*, *emissions_indices*, *concentration_indices*)

Calculate methane lifetime scaling factor.

Parameters

- **emissions** (*np.ndarray*) – Emitted species
- **concentration** (*np.ndarray*) – Species derived from concentration (including EESC)
- **temperature** (*np.ndarray*) – Global mean surface level temperature anomaly
- **baseline_emissions** (*np.ndarray*) – reference, possibly pre-industrial, emissions
- **baseline_concentration** (*np.ndarray*) – reference, possibly pre-industrial, concentration

- **ch4_lifetime_chemical_sensitivity** (*np.ndarray*) – per-species sensitivity of change in CH4 lifetime per unit emission.
- **ch4_lifetime_temperature_sensitivity** (*np.ndarray*) – per-species sensitivity of change in CH4 lifetime per unit concentration increase.
- **emissions_indices** (*np.ndarray of bool*) – Which species indices to use emissions from.
- **concentration_indices** (*np.ndarray of bool*) – Which species indices to use concentration from.

Returns
alpha_ch4 – Scaling factor to baseline methane lifetime.

Return type
np.ndarray

fair.gas_cycle.eesc

Module for calculaing equivalent effective stratospheric chlorine.

fair.gas_cycle.eesc.**calculate_eesc**(*concentration*, *fractional_release*, *cl_atoms*, *br_atoms*, *cfc_11_index*, *halogen_indices*, *br_cl_ratio*)

Calculate equivalent effective stratospheric chlorine.

Parameters

- **concentration** (*np.ndarray*) – concentrations in timestep
- **fractional_release** (*np.ndarray*) – fractional release describing the proportion of available ODS that actually contributes to ozone depletion.
- **cl_atoms** (*np.ndarray*) – Chlorine atoms in each species
- **br_atoms** (*np.ndarray*) – Bromine atoms in each species
- **cfc_11_index** (*int or None*) – array index along SPECIES_AXIS corresponding to CFC-11.
- **halogen_indices** (*list of int*) – provides a mapping of which halogen species corresponds to which index along the SPECIES_AXIS.
- **br_cl_ratio** (*float*) – how much more effective bromine is as an ozone depletor than chlorine.

Returns
eesc_out – equivalent effective stratospheric chlorine

Return type
np.ndarray

Notes

At present, CFC-11 must be provided in the scenario, with species type cfc-11.

fair.gas_cycle.forward

Module for the forward (emissions to concentration) model.

fair.gas_cycle.forward.**step_concentration**(*emissions*, *gasboxes_old*, *airborne_emissions_old*, *alpha_lifetime*, *baseline_concentration*, *baseline_emissions*, *concentration_per_emission*, *lifetime*, *partition_fraction*, *timestep*)

Calculate concentrations from emissions of any greenhouse gas.

Parameters

- **emissions** (*np.ndarray*) – emissions rate (emissions unit per year) in timestep.
- **gas_boxes_old** (*np.ndarray*) – the greenhouse gas atmospheric burden in each lifetime box at the end of the previous timestep.
- **airborne_emissions_old** (*np.ndarray*) – The total airborne emissions at the beginning of the timestep. This is the concentrations above the pre-industrial control. It is also the sum of gas_boxes_old if this is an array.
- **alpha_lifetime** (*np.ndarray*) – scaling factor for *lifetime*. Necessary where there is a state- dependent feedback.
- **baseline_concentration** (*np.ndarray*) – original (possibly pre-industrial) concentration of gas(es) in question.
- **baseline_emissions** (*np.ndarray or float*) – original (possibly pre-industrial) emissions of gas(es) in question.
- **concentration_per_emission** (*np.ndarray*) – how much atmospheric concentrations grow (e.g. in ppm) per unit (e.g. GtCO2) emission.
- **lifetime** (*np.ndarray*) – atmospheric burden lifetime of greenhouse gas (yr). For multiple lifetimes gases, it is the lifetime of each box.
- **partition_fraction** (*np.ndarray*) – the partition fraction of emissions into each gas box. If array, the entries should be individually non-negative and sum to one.
- **timestep** (*float*) – emissions timestep in years.

Notes

Emissions are given in time intervals and concentrations are also reported on the same time intervals: the airborne_emissions values are on time boundaries and these are averaged before being returned.

Where array input is taken, the arrays always have the dimensions of (scenario, species, time, gas_box). Dimensionality can be 1, but we retain the singleton dimension in order to preserve clarity of calculation and speed.

Returns

- **concentration_out** (*np.ndarray*) – greenhouse gas concentrations at the centre of the timestep.
- **gas_boxes_new** (*np.ndarray*) – the greenhouse gas atmospheric burden in each lifetime box at the end of the timestep.

- **airborne_emissions_new** (*np.ndarray*) – airborne emissions (concentrations above pre-industrial control level) at the end of the timestep.

fair.gas_cycle.inverse

Module for deriving emissions from concentration.

fair.gas_cycle.inverse.**unstep_concentration**(*concentration*, *gasboxes_old*, *airborne_emissions_old*, *alpha_lifetime*, *baseline_concentration*, *baseline_emissions*, *concentration_per_emission*, *lifetime*, *partition_fraction*, *timestep*)

Calculate emissions from concentrations of any greenhouse gas.

Parameters

- **concentration** (*np.ndarray*) – greenhouse gas concentration at the centre of the timestep.
- **gas_boxes_old** (*np.ndarray*) – the greenhouse gas atmospheric burden in each lifetime box at the end of the previous timestep.
- **airborne_emissions_old** (*np.ndarray*) – The total airborne emissions at the beginning of the timestep. This is the concentrations above the pre-industrial control. It is also the sum of gas_boxes_old if this is an array.
- **alpha_lifetime** (*np.ndarray*) – scaling factor for *lifetime*. Necessary where there is a state- dependent feedback.
- **baseline_concentration** (*np.ndarray*) – baseline (possibly pre-industrial) concentration of gas in question.
- **baseline_emissions** (*np.ndarray*) – baseline (possibly pre-industrial) emissions of gas in question.
- **concentration_per_emission** (*np.ndarray*) – how much atmospheric concentrations grow (e.g. in ppm) per unit (e.g. GtCO2) emission.
- **lifetime** (*np.ndarray*) – atmospheric burden lifetime of greenhouse gas (yr). For multiple lifetimes gases, it is the lifetime of each box.
- **partition_fraction** (*np.ndarray*) – the partition fraction of emissions into each gas box. If array, the entries should be individually non-negative and sum to one.
- **timestep** (*float*) – emissions timestep in years.

Notes

Emissions are given in time intervals and concentrations are also reported on the same time intervals: the airborne_emissions values are on time boundaries. Therefore it is not actually possible to provide the exact emissions that would reproduce the concentrations without using a slower root-finding mechanism (that was present in v1.4) and will always be half a time step out.

Where array input is taken, the arrays always have the dimensions of (scenario, species, time, gas_box). Dimensionality can be 1, but we retain the singleton dimension in order to preserve clarity of calculation and speed.

Returns

- **emissions_out** (*np.ndarray*) – emissions in timestep.

- **gas_boxes_new** (*np.ndarray*) – the greenhouse gas atmospheric burden in each lifetime box at the end of the timestep.
- **airborne_emissions_new** (*np.ndarray*) – airborne emissions (concentrations above pre-industrial control level) at the end of the timestep.

1.4.9 fair.structure

Module for organising FaIR's structure.

fair.structure.species

Define the types of beasts in FaIR, how they roar, and whether they are friendly.

fair.structure.species.**multiple_allowed = {'aci': False, 'ari': False, 'black carbon': False, 'cfc-11': False, 'ch4': False, 'co2': False, 'co2 afolu': False, 'co2 ffi': False, 'contrails': False, 'eesc': False, 'f-gas': True, 'h2o stratospheric': False, 'land use': False, 'lapsi': False, 'n2o': False, 'nox aviation': False, 'organic carbon': False, 'other halogen': True, 'other slcf': True, 'ozone': False, 'solar': False, 'sulfur': False, 'unspecified': True, 'volcanic': False}**

Whether multiple instances are allowed for each specie type.

fair.structure.species.**species_types = ['co2 ffi', 'co2 afolu', 'co2', 'ch4', 'n2o', 'cfc-11', 'other halogen', 'f-gas', 'sulfur', 'black carbon', 'organic carbon', 'other slcf', 'nox aviation', 'eesc', 'ozone', 'ari', 'aci', 'contrails', 'lapsi', 'h2o stratospheric', 'land use', 'volcanic', 'solar', 'unspecified']**

Species types recognised by FaIR.

fair.structure.species.**valid_input_modes = {'aci': ['calculated', 'forcing'], 'ari': ['calculated', 'forcing'], 'black carbon': ['emissions'], 'cfc-11': ['emissions', 'concentration', 'forcing'], 'ch4': ['emissions', 'concentration', 'forcing'], 'co2': ['emissions', 'calculated', 'concentration', 'forcing'], 'co2 afolu': ['emissions'], 'co2 ffi': ['emissions'], 'contrails': ['calculated', 'forcing'], 'eesc': ['concentration', 'calculated'], 'f-gas': ['emissions', 'concentration', 'forcing'], 'h2o stratospheric': ['calculated', 'forcing'], 'land use': ['calculated', 'forcing'], 'lapsi': ['calculated', 'forcing'], 'n2o': ['emissions', 'concentration', 'forcing'], 'nox aviation': ['emissions'], 'organic carbon': ['emissions'], 'other halogen': ['emissions', 'concentration', 'forcing'], 'other slcf': ['emissions'], 'ozone': ['calculated', 'forcing'], 'solar': ['forcing'], 'sulfur': ['emissions'], 'unspecified': ['forcing'], 'volcanic': ['forcing']}**

Valid run modes for each species.

fair.structure.units

Define species units and conversions.

fair.structure.units.**desired_concentration_units** = {'C2F6': 'ppt', 'C3F8': 'ppt', 'C4F10': 'ppt', 'C5F12': 'ppt', 'C6F14': 'ppt', 'C7F16': 'ppt', 'C8F18': 'ppt', 'CCl4': 'ppt', 'CF4': 'ppt', 'CFC-11': 'ppt', 'CFC-113': 'ppt', 'CFC-114': 'ppt', 'CFC-115': 'ppt', 'CFC-12': 'ppt', 'CH2Cl2': 'ppt', 'CH3Br': 'ppt', 'CH3CCl3': 'ppt', 'CH3Cl': 'ppt', 'CH4': 'ppb', 'CHCl3': 'ppt', 'CO2': 'ppm', 'Equivalent effective stratospheric chlorine': 'ppt', 'HCFC-141b': 'ppt', 'HCFC-142b': 'ppt', 'HCFC-22': 'ppt', 'HFC-125': 'ppt', 'HFC-134a': 'ppt', 'HFC-143a': 'ppt', 'HFC-152a': 'ppt', 'HFC-227ea': 'ppt', 'HFC-23': 'ppt', 'HFC-236fa': 'ppt', 'HFC-245fa': 'ppt', 'HFC-32': 'ppt', 'HFC-365mfc': 'ppt', 'HFC-4310mee': 'ppt', 'Halon-1202': 'ppt', 'Halon-1211': 'ppt', 'Halon-1301': 'ppt', 'Halon-2402': 'ppt', 'N2O': 'ppb', 'NF3': 'ppt', 'SF6': 'ppt', 'SO2F2': 'ppt', 'c-C4F8': 'ppt'}

Desired concentration units for each default specie.

fair.structure.units.**desired_emissions_units** = {'BC': 'Mt BC/yr', 'C2F6': 'kt C2F6/yr', 'C3F8': 'kt C3F8/yr', 'C4F10': 'kt C4F10/yr', 'C5F12': 'kt C5F12/yr', 'C6F14': 'kt C6F14/yr', 'C7F16': 'kt C7F16/yr', 'C8F18': 'kt C8F18/yr', 'CCl4': 'kt CCl4/yr', 'CF4': 'kt CF4/yr', 'CFC-11': 'kt CFC11/yr', 'CFC-113': 'kt CFC113/yr', 'CFC-114': 'kt CFC114/yr', 'CFC-115': 'kt CFC115/yr', 'CFC-12': 'kt CFC12/yr', 'CH2Cl2': 'kt CH2Cl2/yr', 'CH3Br': 'kt CH3Br/yr', 'CH3CCl3': 'kt CH3CCl3/yr', 'CH3Cl': 'kt CH3Cl/yr', 'CH4': 'Mt CH4/yr', 'CHCl3': 'kt CHCl3/yr', 'CO': 'Mt CO/yr', 'CO2': 'Gt CO2/yr', 'CO2 AFOLU': 'Gt CO2/yr', 'CO2 FFI': 'Gt CO2/yr', 'HCFC-141b': 'kt HCFC141b/yr', 'HCFC-142b': 'kt HCFC142b/yr', 'HCFC-22': 'kt HCFC22/yr', 'HFC-125': 'kt HFC125/yr', 'HFC-134a': 'kt HFC134a/yr', 'HFC-143a': 'kt HFC143a/yr', 'HFC-152a': 'kt HFC152a/yr', 'HFC-227ea': 'kt HFC227ea/yr', 'HFC-23': 'kt HFC23/yr', 'HFC-236fa': 'kt HFC236fa/yr', 'HFC-245fa': 'kt HFC245fa/yr', 'HFC-32': 'kt HFC32/yr', 'HFC-365mfc': 'kt HFC365mfc/yr', 'HFC-4310mee': 'kt HFC4310mee/yr', 'Halon-1202': 'kt Halon1202/yr', 'Halon-1211': 'kt Halon1211/yr', 'Halon-1301': 'kt Halon1301/yr', 'Halon-2402': 'kt Halon2402/yr', 'N2O': 'Mt N2O/yr', 'NF3': 'kt NF3/yr', 'NH3': 'Mt NH3/yr', 'NOx': 'Mt NO2/yr', 'NOx aviation': 'Mt NO2/yr', 'OC': 'Mt OC/yr', 'SF6': 'kt SF6/yr', 'SO2F2': 'kt SO2F2/yr', 'Sulfur': 'Mt SO2/yr', 'VOC': 'Mt VOC/yr', 'c-C4F8': 'kt cC4F8/yr'}

Desired emissions units for each specie.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

REFERENCES

BIBLIOGRAPHY

[Cummins2020] Cummins, D. P., Stephenson, D. B., & Stott, P. A. (2020). Optimal Estimation of Stochastic Energy Balance Model Parameters, Journal of Climate, 33(18), 7909-7926.

[Etminan2016] Etminan, M., Myhre, G., Highwood, E.J., Shine, K.P., (2016). Radiative forcing of carbon dioxide, methane, and nitrous oxide: A significant revision of the methane radiative forcing, Geophysical Research Letters, 43, 12,614–12,623

[Geoffroy2013] Geoffroy, O., Saint-Martin, D., Bellon, G., Voldoire, A., Olivié, D. J. L., & Tytéca, S. (2013). Transient Climate Response in a Two- Layer Energy-Balance Model. Part II: Representation of the Efficacy of Deep-Ocean Heat Uptake and Validation for CMIP5 AOGCMs, Journal of Climate, 26(6), 1859-1876

[Leach2021] Leach, N. J., Jenkins, S., Nicholls, Z., Smith, C. J., Lynch, J., Cain, M., Walsh, T., Wu, B., Tsutsui, J., and Allen, M. R. (2021). FaIRv2.0.0: a generalized impulse response model for climate uncertainty and future scenario exploration. Geoscientific Model Development, 14, 3007–3036

[Meinshausen2020] Meinshausen, M., Nicholls, Z.R.J., Lewis, J., Gidden, M.J., Vogel, E., Freund, M., Beyerle, U., Gessner, C., Nauels, A., Bauer, N., Canadell, J.G., Daniel, J.S., John, A., Krummel, P.B., Luderer, G., Meinshausen, N., Montzka, S.A., Rayner, P.J., Reimann, S., Smith, S.J., van den Berg, M., Velders, G.J.M., Vollmer, M.K., Wang, R.H.J. (2020). The shared socio-economic pathway (SSP) greenhouse gas concentrations and their extensions to 2500, Geoscientific Model Development, 13, 3571–3605.

[Millar2017] Millar, R. J., Nicholls, Z. R., Friedlingstein, P., and Allen, M. R. (2017) A modified impulse-response representation of the global near-surface air temperature and atmospheric concentration response to carbon dioxide emissions. Atmospheric Chemistry and Physics, 17, 7213–7228.

[Myhre1998] Myhre, G., Highwood, E.J., Shine, K. Stordal, F. (1998). New estimates or radiative forcing due to well mixed greenhouse gases. Geophysical Research Letters, 25 (14), 2715-2718.

[Skeie2020] Skeie, R.B., Myhre, G., Hodnebrog, Ø., Cameron-Smith, P.J., Deushi, M., Hegglin, M.I., Horowitz, L.W., Kramer, R.J., Michou, M., Mills, M.J., Olivié, D.J., Connor, F.M., Paynter, D., Samset, B.H., Sellar, A., Shindell, D., Takemura, T., Tilmes, S., Wu, T., 2020. Historical total ozone radiative forcing derived from CMIP6 simulations, npj Climate and Atmospheric Science, 3, 1–10.

[Stevens2015] Stevens, B. (2015). Rethinking the Lower Bound on Aerosol Radiative Forcing, Journal of Climate, 28(12), 4794-4819.

[Thornhill2021a] Thornhill, G.D., Collins, W.J., Kramer, R.J., Olivié, D., Skeie, R.B., O'Connor, F.M., Abraham, N.L., Checa-Garcia, R., Bauer, S.E., Deushi, M., Emmons, L.K., Forster, P.M., Horowitz, L.W., Johnson, B., Keeble, J., Lamarque, J.-F., Michou, M., Mills, M.J., Mulcahy, J.P., Myhre, G., Nabat, P., Naik, V., Oshima, N., Schulz, M., Smith, C.J., Takemura, T., Tilmes, S., Wu, T., Zeng, G., Zhang, J. (2021). Effective radiative forcing from emissions of reactive gases and aerosols – a multi-model comparison, Atmospheric Chemistry and Physics, 21, 853–874

[Tsutsui2017] Tsutsui (2017): Quantification of temperature response to CO2 forcing in atmosphere–ocean general circulation models. Climatic Change, 140, 287–305

PYTHON MODULE INDEX

f

INDEX

G

I

L

M

R

S

T

U

V